Second Revised & Updated Edition

DATA STRUCTURES THROUGH C IN DEPTH

S. K. Srivastava
Deepali Srivastava

BPB PUBLICATIONS

BPB PUBLICATIONS

Meet Asia's Largest Publisher of Computer & Electronic Books

Ask anyone even remotely associated with the computer: What is BPB? You'll get the answer: the reservoir of knowledge. Welcome to the family of BPB Publications! A family with about 60 million people residing in diverse parts of the Indian subcontinent and even farther than that. A family with computer literate people who have achieved this acumen and expertise by the perpetual quest for excellence by BPB. With such adornments as 6000 publications, over 90 million books sold over the years and about the same number of reader base, it is a matter of no surprise that BPB titles are prescribed as standard courseware at most of the leading schools, institutes, and universities in India.

#1
Publisher of Computer Books

62
Years of Excellence

OVER
90
Million Books Sold Worldwide

For 62 years, BPB has been a friend, philosopher and guide for people like software engineers and programmers, developers, hardware technicians, IT professionals and students who have made things happen in the IT world. BPB has proved its success in the field of computer education, and anyone who has graduated in Computer or Electronics has graduated with BPB books. Our chairman Shri G.C. Jain has been honoured with the PADMASHREE award by the Hon'ble President of India for his great contribution in spreading the IT education in India. There is no doubt that BPB has reached out and changed the lives of millions of people. All over India and Asia, it is its endeavour to do more.

www.bpbonline.com

- Computer Books
- E Books
- Video Courses

Data Structures Through C in Depth

Second Revised & Updated Edition

by
S.K. Srivastava
Deepali Srivastava

BPB PUBLICATIONS
20 Ansari Road, Darya Ganj, New Delhi-110002

REPRINTED 2021
SECOND REVISED & UPDATED EDITION 2011
Copyright © 2011 BPB Publications, INDIA
ISBN 10: 81-7656-741-8
ISBN 13: 978-81-7656-741-1

All Rights Reserved. No part of this publication can be stored in a retrieval system or reproduced in any form or by any means without the prior written permission of the publishers.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The Author and Publisher of this book have tried their best to ensure that the programs, procedures and functions described in the book are correct. However, the author and the publishers make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in the book. The Author & Publisher shall not be liable in any event of any damages, incidental or consequential, in connection with, or arising out of the furnishing, performance or use of these programs, procedures and functions, Product name mentioned are used for dentification purposes only and may be trademarks of their respective companies.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi- 110002
Ph: 23254990 /23254991

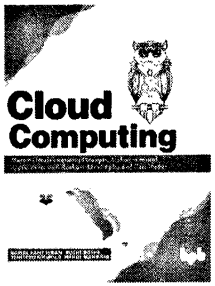
DECCAN AGENCIES
4-3-329, Bank Street,
HYDERABAD-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
DELHI-110006
Ph: 23861747

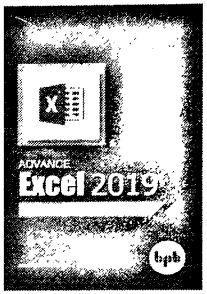
Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002, and Printed by Adinath Printer, New Delhi.

To our Parents
and
our lovely daughter
DEVANSHI

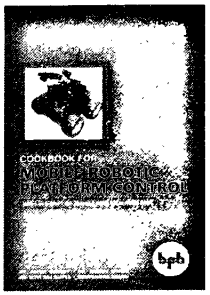


Cloud Computing
Author: Kant,K/Doshi,R
ISBN 9789388511407

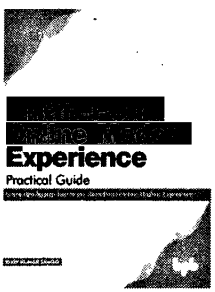
Writing Quality Research Papers
Author: Dr.Singh,P/ Dr. Khan,B
ISBN 9789388176903



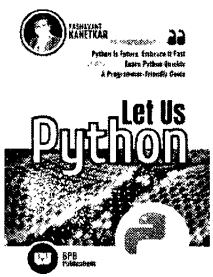
Advance Excel 2019 Training Guide
Author: Nigam,M
ISBN 9789388176675



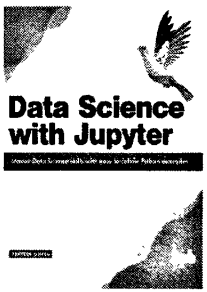
Cookbook for Mobile Robotic Platform Control
Author: Dr. Gehlot,A/ Dr. Singh,R/ Dr. Gupta,LR/ Singh,B
ISBN 9789388511674



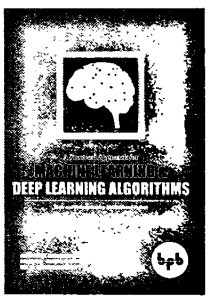
SharePoint Online Modern Experience Practical Guide
Author: Sahoo,BK
ISBN 9789388511575



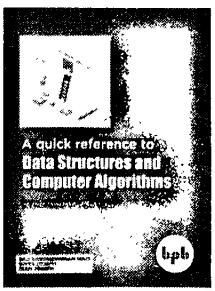
Let Us Python
Author: Kanetkar,Y
ISBN 9789388511568



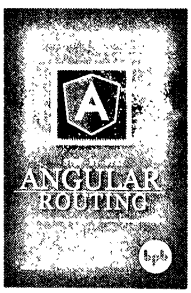
Data Science with Jupyter
Author: Gupta,P
ISBN 9789388511377



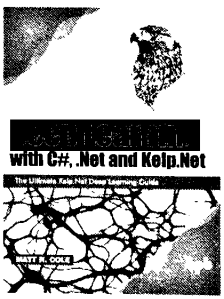
A Practical Approach for Machine Learning & Deep Learning Algorithms
Author: Pandey,AK / Rathore,PS/ Dr. Balamurgan,S
ISBN 9789388511131



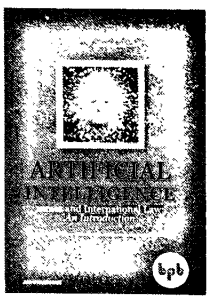
A Quick Reference to Data Structure and Computer Algorithms
Author: Nair,RR/ Joseph,D/ Joseph,A
ISBN 9789388176583



Step-by-Step Angular Routing
Author: Goel,N
ISBN 9789388511667



Deep Learning with C#, .Net and Kelp.Net
Author: Cole,MR
ISBN 9789388511018



Artificial Intelligence Ethics and International Law: An Introduction
Author: Abhivardhan
ISBN 9789388511629

Available on **www.bpbonline.com** and all leading book stores.

#1 Publisher of Computer Books | 62 Years of Excellence | OVER 90 Million Books Sold Worldwide

Preface

The study of data structures is an integral part of any course in Computer Science. This book focuses on various data structures and their implementation in C language. The prerequisite for this book is a basic knowledge of programming in C language. We assume that the reader is familiar with the structured programming concepts and syntax of C language. This book can be used by students for self study as the concepts are explained in step-by-step manner followed by clear and easy to comprehend complete programs. We have followed a figure-oriented approach in this book, so there are numerous figures and tables throughout the book to illustrate the working of algorithms. This is the second edition of the book, most of the topics have been thoroughly rewritten and many new topics have been added but we have tried to retain the presentation style and simplicity of the previous edition. A brief synopsis of the book follows-

- Chapter 1 is an introduction to data structures, abstract data types, algorithms and efficiency.
- Chapter 2 covers fundamental topics of C, like arrays, pointers and structures. It helps in refreshing your knowledge of C, which is essential for understanding the implementations of various algorithms given in the book.
- Chapter 3 introduces the data structure linked list and discusses its different variations.
- Chapter 4 covers the data structures stacks and queues and their different implementations. Applications of stacks are discussed in detail.
- Chapter 5 discusses the concept of recursion. Many students have problem in understanding recursive solutions, so we decided to add a chapter on recursion in the new edition. This chapter explains recursion by tracing different recursive examples.
- Chapter 6 deals with an important data structure Trees. Traversal, insertion and deletion in binary search trees, threaded trees, AVL trees, red black trees, B-trees are explained in detail with various examples.
- Chapter 7 introduces the data structure Graphs. It covers different implementations of graphs and different algorithms related to graph data structure.
- Chapter 8 covers different sorting methods with their analysis.
- Chapter 9 describes various searching and hashing techniques.
- Chapter 10 discusses storage management concepts.

The exercises given at the end of chapters are thoughtful and promote deep understanding of the subject. They help in testing your understanding of the concepts and in applying those concepts, we recommend that readers go through all the exercises and try to solve them before looking at the solutions. Solutions for all the exercises are provided in the book and the companion disc. The disc contains all the programs given in the book. We have included some 'demo' programs in the disc which demonstrate the stepwise working of a program. You can run these programs, input your data and see for yourselves how these algorithms work.

We would like to express our gratitude to our teachers who introduced us to the world of computers, programming and data structures. We are thankful to our publishers for considering our work and letting this book see the light of day. We would like to thank our family and friends for their continuous love and support. Our thanks also go to all our readers of first edition who sent us mails of appreciation and constructive criticism.

We look forward to receiving many more mails from our readers and hope that this book proves helpful to those whom it is meant for.

Readers are welcome to send their queries, suggestions for improvement or information about any error in the book by sending email to us at -

bpb@vsnl.com

Suresh Kumar Srivastava
Deepali Srivastava

CONTENTS

Chapter 1. Introduction 1

1.1 Data Type 1
1.2 Abstract data types 1
1.3 Data structures 2
1.3.1 Linear and Non linear data structures 3
1.3.2 Static and dynamic data structures 3
1.4 Algorithms 4
1.4.1 Greedy algorithm 4
1.4.2 Divide and conquer algorithm 4
1.4.3 Backtracking 4
1.4.4 Randomized algorithms 4
1.5 Analysis of algorithms 4
1.5.1 Big O notation 6
1.5.1.1 Rules for O notation 6.

Chapter 2. Arrays, Pointers and Structures 10

2.1 Arrays 10
2.1.1 One Dimensional Array 10
2.1.1.1 Declaration of 1-D Array 10
2.1.1.2 Accessing 1-D Array Elements 11
2.1.1.3 Processing 1-D Arrays 11
2.1.1.4 Initialization of 1-D Array 12
2.1.1.5 1-D Arrays and Functions 14
2.1.1.5.1 Passing Individual Array Elements to a Function 14
2.1.1.5.2 Passing whole 1-D Array to a Function 14
2.1.2 Two Dimensional Arrays 15
2.1.2.1 Declaration and Accessing Individual Elements of a 2-D array 15
2.1.2.2 Processing 2-D Arrays 15
2.1.2.3 Initialization of 2-D Arrays 16
2.2 Pointers 18
2.2.1 Declaration of a Pointer Variable 18
2.2.2 Assigning Address to a Pointer Variable 19
2.2.3 Dereferencing Pointer Variables 19
2.2.4 Pointer to Pointer 20
2.2.5 Pointers and One Dimensional Arrays 21
2.2.6 Pointers and Functions 23
2.2.7 Returning More Than One Value from a Function 24
2.2.8 Function Returning Pointer 25
2.2.9 Passing a 1-D Array to a Function 26
2.2.10 Array of Pointers 27
2.3 Dynamic Memory Allocation 27
2.3.1 malloc( ). 28
2.3.2 calloc( ) 29
2.3.3 realloc( ) 29

2.3.4 free(-) 30
2.4 Structure 31
2.4.1 Defining a Structure 31
2.4.2 Declaring Structure Variables 32
2.4.2.1 With Structure Definition 32
2.4.2.2 Using Structure Tag 32
2.4.3 Initialization of Structure Variables 33
2.4.4 Accessing Members of a Structure 33
2.4.5 Assignment of Structure Variables 34
2.4.6 Array of Structures 34
2.4.7 Arrays within Structures 35
2.4.8 Nested Structures (Structure within Structure) 36
2.4.9 Pointers to Structures 38
2.4.10 Pointers within Structures 39
2.4.11 Structures and Functions 39
2.4.11.1 Passing Structure Members as Arguments 39
2.4.11.2 Passing Structure Variable as Argument 40
2.4.11.3 Passing Pointers to Structures as Arguments 40
2.4.11.4 Returning a Structure Variable from Function 41
2.4.11.5 Returning a Pointer to Structure from a Function 42
2.4.11.6 Passing Array of Structures as Argument 42
2.4.11.7 Self Referential Structures 43
Exercise 43

Chapter 3. Linked Lists 48

3.1 Single Linked list 48
3.1.1 Traversing a Single Linked List 51
3.1.2 Searching in a Single Linked List 53
3.1.3 Insertion in a Single Linked List 53
3.1.3.1 Insertion at the beginning of the list 53
3.1.3.2 Insertion in an empty list 54
3.1.3.3 Insertion at the end of the list 55
3.1.3.4 Insertion in between the list nodes 55
3.1.3.4.1 Insertion after a node 56
3.1.3.4.2 Insertion before a node 57
3.1.3.4.3 Insertion at a given position 58
3.1.4 Creation of a Single Linked List 58
3.1.5 Deletion in a Single Linked List 59
3.1.5.1 Deletion of first node 59
3.1.5.2 Deletion of the only node 59
3.1.5.3 Deletion in between the list nodes 60
3.1.5.4 Deletion at the end of the list 60
3.1.6 Reversing a Single Linked List 61
3.2 Doubly linked list 63
3.2.1 Traversing a doubly linked List 65
3.2.2 Insertion in a doubly linked List 65
3.2.2.1 Insertion at the beginning of the list 65
3.2.2.2 Insertion in an empty list 66
3.2.2.3 Insertion at the end of the list 66
3.2.2.4 Insertion in between the nodes 67

3.2.3 Creation of List 69
3.2.4 Deletion from doubly linked list 69
3.2.4.1 Deletion of the first node 69
3.2.4.2 Deletion of the only node 70
3.2.4.3 Deletion in between the nodes 70
3.2.4.4 Deletion at the end of the list 70
3.2.5 Reversing a doubly linked list 72
3.3 Circular linked list 72
3.3.1 Traversal in circular linked list 74
3.3.2 Insertion in a circular Linked List 75
3.3.2.1 Insertion at the beginning of the list 75
3.3.2.2 Insertion in an empty list 75
3.3.2.3 Insertion at the end of the list 76
3.3.2.4 Insertion in between the nodes 76
3.3.3 Creation of circular linked list 77
3.3.4 Deletion in circular linked list 77
3.3.4.1 Deletion of the first node 77
3.3.4.2 Deletion of the only node 77
3.3.4.3 Deletion in between the nodes 78
3.3.4.4 Deletion at the end of the list 78
3.4 Linked List with Header Node 79
3.5 Sorted linked list 83
3.6 Sorting a Linked List 86
3.6.1 Selection Sort by exchanging data 87
3.6.2 Bubble Sort by exchanging data 88
3.6.3 Selection Sort by rearranging links 90
3.6.4 Bubble sort by rearranging links 92
3.7 Merging 93
3.8 Concatenation 97
3.9 Polynomial arithmetic with linked list 98
3.9.1 Creation of polynomial linked list 101
3.9.2 Addition of 2 polynomials 101
3.9.3 Multiplication of 2 polynomials 103
3.10 Comparison of Array lists and Linked lists 104
3.10.1 Advantages of linked lists 104
3.10.2 Disadvantages of linked lists 105
Exercise 105

Chapter 4. Stacks and Queues 108

4.1 Stack 108
4.1.1 Array Implementation of Stack 109
4.1.2 Linked List Implementation of Stack 111
4.2 Queue 114
4.2.1 Array Implementation of Queue 114
4.2.2 Linked List implementation of Queue 117
4.3 Circular Queue 122
4.4 Deque 126
4.5 Priority Queue 130
4.6 Applications of stack 132
4.6.1 Reversal of string 133

4.6.2 Checking validity of an expression containing nested parentheses 133
4.6.3 Function calls 136
4.6.4 Polish Notation 137
4.6.4.1 Converting infix expression to postfix expression using stack 139
4.6.4.2 Evaluation of postfix expression using stack 141
Exercise 145

Chapter 5. Recursion 147

5.1 Writing a recursive function 147
5.2 Flow of control in Recursive functions 148
5.3 Winding and unwinding phase 150
5.4 Examples of Recursion 150
5.4.1 Factorial 150
5.4.2 Summation of numbers from 1 to n 152
5.4.3 Displaying numbers from 1 to n 152
5.4.4 Display and Summation of series 154
5.4.5 Sum of digits of an integer and displaying an integer as sequence of characters 155
5.4.6 Base conversion 156
5.4.7 Exponentiation of a float by a positive integer 156
5.4.8 Prime Factorization 157
5.4.9 Greatest Common Divisor 158
5.4.10 Fibonacci Series 158
5.4.11 Checking Divisibility by 9 and 11 159
5.4.12 Tower of Hanoi 160
5.5 Recursive Data structures 163
5.5.1 Strings and recursion 163
5.5.2 Linked lists and recursion 164
5.6 Implementation of Recursion 166
5.7 Recursion vs. Iteration 166
5.8 Tail recursion 167
5.9 Indirect and direct Recursion 169
Exercise 169

Chapter 6. Trees 176

6.1 Terminology 176
6.2 Definition of Tree 178
6.3 Binary Tree 178
6.4 Strictly Binary Tree 180
6.5 Extended Binary Tree 181
6.6 Full binary tree 182
6.7 Complete Binary Tree 182
6.8 Representation of Binary Trees in Memory 183
6.8.1 Array Representation of Binary trees 183
6.8.2 Linked Representation of Binary Trees 184
6.9 Traversal in Binary Tree 186
6.9.1 Non recursive traversals for binary tree 188
6.9.1.1 Preorder Traversal 189
6.9.1.2 Inorder Traversal 190

6.9.1.3 Postorder Traversal 192
6.9.2 Level order traversal 194
6.9.3 Creation of binary tree from inorder and preorder traversals 195
6.9.4 Creation of binary tree from inorder and postorder traversals 197
6.10 Height of Binary tree 200
6.11 Expression tree 201
6.12 Binary Search Tree 202
6.12.1 Traversal in Binary Search Tree 203
6.12.2 Searching in a Binary Search Tree 203
6.12.3 Finding nodes with Minimum and Maximum key 205
6.12.4 Insertion in a Binary Search Tree 205
6.12.5 Deletion in a Binary Search Tree 208
6.13 Threaded Binary Tree 214
6.13.1 Finding inorder successor of a node in in-threaded tree 217
6.13.2 Finding inorder predecessor of a node in in-threaded tree 217
6.13.3 Inorder Traversal of in-threaded binary tree 217
6.13.4 Preorder traversal of in-threaded binary tree 218
6.13.5 Insertion and deletion in threaded binary tree 218
6.13.5.1 Insertion 218
6.13.5.2 Deletion 221
6.13.6 Threaded tree with Header node 224
6.14 AVL Tree 225
6.14.1 Searching and Traversal in AVL tree 227
6.14.2 Tree Rotations 227
6.14.2.1 Right Rotation 227
6.14.2.2 Left Rotation 229
6.14.3 Insertion in an AVL tree 231
6.14.3.1 Insertion in left Subtree 236
6.14.3.2 Insertion in Right Subtree 241
6.14.4 Deletion in AVL tree 247
6.14.4.1 Deletion from Left Subtree 248
6.14.4.2 Deletion from Right Subtree 252
6.15 Red Black Trees 258
6.15.1 Searching 260
6.15.2 Insertion 260
6.15.3 Deletion 264
6.16 Heap 277
6.16.1 Insertion in Heap 279
6.16.2 Deletion 282
6.16.3 Building a heap 284
6.16.4 Selection algorithm 286
6.16.5 Implementation of Priority Queue 286
6.17 Weighted path length 286
6.18 Huffman Tree 287
6.18.1 Application of Huffman tree : Huffman Codes 289
6.19 General Tree 291
6.20 Multiway search tree 292
6.21 B-tree 293
6.21.1 Searching in B-tree 294
6.21.2 Insertion in B-tree 295
6.21.3 Deletion in B-tree 297

6.21.3.1 Deletion from leaf node 297
6.21.3.1.1 If node has more than MIN keys. 297
6.21.3.1.2 If node has MIN keys 298
6.21.3.2 Deletion from non leaf node. 301
6.21.4 Searching 304
6.21.5 Insertion 305
6.21.6 Deletion 311
6.22 B+ tree 318
6.22.1 Searching 319
6.21.2 Insertion 319
6.21.3 Deletion 320
6.22 Digital Search Trees 321
Exercise 322

Chapter 7. Graphs 326

7.1 Undirected Graph 326
7.2 Directed Graph 326
7.3 Graph Terminology 326
7.4 Connectivity in Undirected Graph 329
7.4.1 Connected Graph 329
7.4.2 Connected Components 330
7.4.3 Bridge 330
7.4.4 Articulation point 330
7.4.5 Biconnected graph 331
7.4.6 Biconnected components 332
7.5 Connectivity in Directed Graphs 332
7.5.1 Strongly connected Graph 332
7.5.2 Strongly connected components 332
7.5.3 Weakly connected 333
7.6 Tree 333
7.7 Forest 334
7.8 Spanning tree 334
7.9 Spanning Forest 334
7.10 Representation of Graph 335
7.10.1 Adjacency Matrix 335
7.10.2 Adjacency List 339
7.10.2.1 Vertex insertion 340
7.10.2.2 Edge insertion 340
7.10.2.3 Edge deletion 341
7.10.2.4 Vertex deletion 341
7.11 Transitive closure of a directed graph and Path Matrix 346
7.11.1 Computing Path matrix from powers of adjacency matrix 346
7.11.2 Warshall's Algorithm 349
7.12 Traversal 353
7.12.1 Breadth First Search 353
7.12.1.1 Implementation of Breadth First Search using queue 355
7.12.2 Depth First Search 364
7.12.2.1 Implementation of Depth First Search using stack 366
7.12.2.2 Recursive Implementation of Depth First Search 370
7.12.2.3 Classification of Edges in DFS 373

7.12.2.4 Strongly connected graph and strongly connected components 376
7.13 Shortest Path Problem 377
7.13.1 Dijkstra's Algorithm 378
7.13.2 Bellman Ford Algorithm 387
7.13.3 Modified Warshall's Algorithm (Floyd's Algorithm) 392
7.14 Minimum spanning tree 398
7.14.1 Prim's Algorithm 398
7.14.2 Kruskal's Algorithm 405
7.15 Topological Sorting 410
Exercise 414

Chapter 8. Sorting 417

8.1 Sort Order 418
8.2 Types of Sorting 418
8.3 Sort Stability 418
8.4 Sort by Address(Indirect Sort) 419
8.5 In place Sort 420
8.6 Sort pass 420
8.7 Sort Efficiency 421
8.8 Selection Sort 421
8.8.1 Analysis of Selection Sort 423
8.9 Bubble Sort 424
8.9.1 Analysis of Bubble Sort 426
8.9.1.1 Data in sorted order 426
8.9.1.2 Data in reverse sorted order 427
8.9.1.3 Data in random order 427
8.10 Insertion Sort 427
8.10.1 Analysis of Insertion sort 429
8.10.1.1 Data in sorted order 430
8.10.1.2 Data in reverse sorted order 430
8.10.1.3 Data in random order 430
8.11 Shell Sort (Diminishing Increment Sort) 431
8.11.1 Analysis of Shell Sort 434
8.12 Merge Sort 434
8.12.1 Top Down Merge Sort (Recursive) 436
8.12.2 Analysis of Merge Sort 439
8.12.3 Bottom Up Merge Sort(Iterative) 439
8.12.4 Merge Sort for linked List 441
8.12.5 Natural Merge Sort 444
8.13 Quick Sort (Partition Exchange Sort) 444
8.13.1 Analysis of Quick Sort 449
8.13.2 Choice of pivot in Quick Sort 450
8.13.3 Duplicate elements in quick sort 450
8.14 Binary tree sort 451
8.14.1 Analysis of Binary Tree Sort 454
8.15 Heap Sort 455
8.15.1 Analysis of Heap Sort 458
8.16 Radix Sort 458
8.16.1 Analysis of Radix Sort 462
8.17 Address Calculation Sort 462

8.17.1 Analysis of Address Calculation Sort 465
Exercise 470

Chapter 9. Searching and Hashing 472

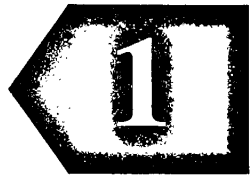
9.1 Sequential Search (Linear search) 472
9.2 Binary Search 473
9.3 Hashing 476
9.3.1 Hash functions 478
9.3.1.1 Truncation (or Extraction) 478
9.3.1.2 Midsquare Method 479
9.3.1.3 Folding Method 479
9.3.1.4 Division Method(Modulo-Divison) 479
9.3.2 Collision Resolution 480
9.3.2.1 Open Addressing (Closed Hashing) 480
9.3.2.1.1 Linear Probing 480
9.3.2.1.2 Quadratic Probing 481
9.3.2.1.3 Double Hashing 482
9.3.2.1.4 Deletion in open addressed tables 484
9.3.2.1.5 Implementation of open Addressed tables 484
9.3.2.2 Separate chaining 487
9.3.3 Bucket Hashing 490
Exercise 491

Chapter 10. Storage Management 492

10.1 Sequential Fit Methods 493
10.1.1 First Fit 493
10.1.2 Best Fit method 493
10.1.3 Worst Fit Method 493
10.2 Fragmentation 494
10.3 Freeing memory 494
10.4 Boundary tag method 495
10.5 Buddy Systems 498
10.5.1 Binary Buddy System 498
10.5.2 Fibonacci Buddy System 502
10.6 Compaction 506
10.7 Garbage collection 506
10.7.1 Reference counting 506
10.7.2 Mark and Sweep 507
Exercise 507

Solutions 509
Index 522

Introduction



Generally any problem that has to be solved by the computer involves the use of data. If data is arranged in some systematic way then it gets a structure and becomes meaningful. This meaningful or processed data is called information. It is essential to manage data in such a way so that it can produce information. There can be many ways in which data may be organized or structured. To provide an appropriate structure to your data you need to know about data structures. Data structures can be viewed as a systematic way to organize data so that it can be used efficiently. The choice of proper data structure can greatly affect the efficiency of our program.

1.1 Data Type

A data type defines a domain of allowed values and the operations that can be performed on those values. For example in C, int data type can take values in a range and operations that can be performed are addition, subtraction, multiplication, division, bitwise operations etc. Similarly the data type float can take values in a particular range and operations allowed are addition, subtraction, multiplication, division etc (% operation, bitwise operations are not allowed). These are built-in data types or primitive data types and the values and operations for them are defined in the language.

If an application needs to use a data type other than the primitive data types of the language, i.e. a data type for which values and operations are not defined in the language itself, then it is programmer's responsibility to specify the values and operations for that data type and implement it. For example there is no built in type for dates in C, and if we need to process dates we have to define and implement a data type for date.

1.2 Abstract data types

Abstract data type is a mathematical model or concept that defines a data type logically. It specifies a set of data and collection of operations that can be performed on that data. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of a data type need not know how the data type is implemented, for example we have been using int, float, char data types only with the knowledge of values that they can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

List ADT

A list contains elements of same type arranged in sequential order and following operations can be performed on the list.

Initialize( ) - Initialize the list to be empty.

get( ) - Return an element from the list at any given position.

insert( ) - Insert a new element at any position of the list.
remove( ) - Remove the first occurrence of any element from a non empty list.
removeAt( ) - Remove the element at a specified location from a non empty list.
replace( ) - Replace an element at any position by another element.
size( ) - Return the number of elements in the list.
isEmpty( ) - Return true if the list is empty, otherwise return false.
isFull( ) - Return true if the list is full, otherwise return false.

Stack ADT

A stack contains elements of same type arranged in sequential order and following operations can be performed on the stack.

Initailize( ) - Initialize the stack to be empty.
Push( ) - Insert an element at one end of the stack called top.
Pop( ) - Remove and return the element at the top of the stack, if it is not empty.
Peek( ) - Return the element at the top of the stack without removing it, if the stack is not empty.
size( ) - Return the number of elements in the stack.
isEmpty( ) - Return true if stack is empty, otherwise return false.
isFull( ) - Return true if no more elements can be pushed, otherwise return false.

Queue ADT

A queue contains elements of same type arranged in sequential order and following operations can be performed on the queue.

Initailize( ) - Initialize the queue to be empty.
Enqueue( ) - Insert an element at the end of queue.
Dequeue( ) - Remove and return the first element of queue, if queue is not empty.
Peek( ) - Return first element of the queue without removing it, if queue is not empty.
size( ) - Return the number of elements in the queue.
isEmpty( ) - Return true if queue is empty, otherwise return false.
isFull( ) - Return true if no more elements can be inserted, otherwise return false.

From these definitions we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example the list ADT can be implemented using arrays, or single linked list or double linked list. Similarly stack ADT and queue ADT can also be implemented using arrays or linked lists. The representation and implementation details of these ADTs are in chapters 3 and 4.
The different implementations of ADT are compared for time and space efficiency and the implementation best suited for the requirement of the user is used. For example if someone wants to use a list in a program which involves lots of insertions and deletions from the list, then it is better to use the linked list implementation of list.

1.3 Data structures

Data structure is a programming construct used to implement an ADT. It is the physical implementation of ADT. All operations specified in ADT are implemented through functions in the physical implementation. A data structure that is implementing an ADT consists of collection of variables for storing the data specified in the ADT and the algorithms for implementing the operations specified in ADT.

ADT is the logical view of data and the operations to manipulate the data while Data structure is the actual representation of data and the algorithms to manipulate the data.

We can say that ADT is a specification language for data structures. ADT is a logical description while Data Structure is concrete i.e. an ADT tells us **what** is to be done and data structures tells us **how** to do it. The actual storage or representation of data and implementation of algorithms is done in data structures.

A program that uses a data structure is generally called a client, and the program that implements it is known as the implementation. The specification of ADT is called interface and it is the only thing visible to the client programs that use data structure. The client programs view data structure as ADT, i.e. they have access only to the interface; the way data is represented and operations are implemented is not visible to the client. For example if someone wants to use a stack in the program, he can simply use push and pop operations without any knowledge of how they are implemented. Some examples of clients of stack ADT are programs of balanced parentheses, infix to postfix, postfix evaluation.

We may change the representation or algorithm but the client code will not be affected as long as the ADT interface remains the same. For example if the implementation of stack is changed from array to linked list, the client program should work in the same way. This helps the user of a data structure focus on his program rather than going into the details of the data structure.

Data structures can be nested i.e. a data structure may be made up of other data structures which may be of primitive types or user defined types. Some of the advantages of data structures are-

(i) Efficiency - Proper choice of data structures makes our program efficient. For example suppose we have some data and we need to organize it properly and perform search operation. If we organize our data in an array, then we will have to search sequentially element by element. If item to be searched is present at last then we will have to visit all the elements before reaching that element. So use of an array is not very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables. Different data structures give different efficiency.

(ii) Reusability - Data structures are reusable, i.e. once we have implemented a particular data structure we can use it in any other place or requirement. Implementations of data structures can be compiled into libraries which can be used by different clients.

(iii) Abstraction - We have seen that a data structure is specified by an ADT which provides a level of abstraction. The client program uses the data structure through the interface only without getting into the implementation details.

Some common operations that are performed on Data structures are-
(i) Insertion
(ii) Deletion
(iii) Traversal
(iv) Search

There can be other operations also and the details of these operations depend on different data structures.

1.3.1 Linear and Non linear data structures

A data structure is linear if all the elements are arranged in a linear order. In a linear data structure, each element has only one successor and only one predecessor. The only exceptions are the first and the last elements; first element does not have a predecessor and last element does not have a successor. The examples of linear data structures are array, string, linked list, stack, and queue.

In a non linear data structure, there is no linear order in the arrangement of the elements. The examples of non linear data structures are trees and graphs.

1.3.2 Static and dynamic data structures

In a static data structure, the memory is allocated at compilation time only. Therefore, the maximum size is fixed and it can't be changed at run time. Static data structures allow fast access to elements but insertion and deletion is expensive. Array is an example of static data structure.

In a dynamic data structure, the memory is allocated at run time. Therefore, these data structures have flexible size. Dynamic data structures allow fast insertion and deletion of elements but access to elements is slow. Linked list is an example of dynamic data structure.

1.4 Algorithms

An algorithm is a procedure having well defined steps for solving a particular problem. The data stored in the data structures is manipulated by using different algorithms, so the study of data structures includes the study of algorithms. Some of the common approaches of algorithm design are-

1.4.1 Greedy algorithm

A greedy algorithm works by taking a decision that appears best at the moment, without thinking about the future. The decision once taken is never reconsidered. This means that a local optimum is chosen at every step in hope of getting a global optimum at the end. It is not necessary that a greedy algorithm will always produce an optimal solution. Some examples where greedy approach produces optimal solutions are Dijkstra's algorithm for single source shortest paths, Prim's and Kruskal's algorithm for minimum spanning tree, and Huffmann algorithm.

1.4.2 Divide and conquer algorithm

A divide and conquer algorithm solves a problem by dividing it into smaller and similar subproblems. The solutions of these smaller problems are then combined to get the solution of the given problem. The examples of divide and conquer algorithms are merge sort, quick sort, binary search.

1.4.3 Backtracking

In some problems we have several options, where any one might lead to the solution. We will take an option and try, and if we do not reach the solution, we will undo our action and select another one. The steps are retraced if we do not reach the solution; it is a trial and error process. An example of backtracking is the Eight Queens problem.

1.4.4 Randomized algorithms

In a randomized algorithm, random numbers are used to make some decisions. The running time of such algorithms depends on the input as well as the random numbers that are generated. The running time may vary for the same input data. An example of randomized algorithm is a version of quick sort where a random number is chosen as the pivot.

1.5 Analysis of algorithms

There may be many algorithms for solving any problem and obviously we would like to use the most efficient one. Analysis of algorithms is required to compare these algorithms and recognize the best one. Algorithms are generally analyzed on their time and space requirements.

One way of comparing algorithms is to compare the exact running time of all algorithms. But the running time is dependent on the language and machine used for implementing the algorithm. Even if the machine and language are kept same, calculation of exact time would be very difficult as it would require the count of instructions executed by the hardware and the time taken to execute each instruction. So the time efficiency is not measured in time units like seconds or microseconds.

The running time generally depends on the size of input, for example any sorting algorithm will take less time to sort 10 elements and more time for 100000 elements. So the time efficiency is generally expressed in terms of size of input. If the size of input is n, then f(n) which is a function of n denotes the time complexity. Thus to compare any two algorithms we will find out this function for both algorithms and then compare the rate of growth of these two functions. It is important to compare the rates of growth because an algorithm may seem better for small input but as the input becomes large it may take more time than others.

The function f(n) may be found out by identifying some key operations in the algorithm which account for most of the running time. Other operations are not counted as they take very little time as compared to these key operations and not executed more often than the key operations. For example, in searching we may count the number of comparisons and in sorting we may count the swaps in addition to comparisons. We are interested only in the growth rate of functions so the exact computation of f(n) is not necessary.

Let us take an example where time complexity is given by the following function-
$f(n) = 5n^2 + 6n + 12$
If n=10
% of running time due to the term $5n^2$: (500 / (500 + 60 +12)) *100 = 87.41 %
% of running time due to the term 6n : (60 / (500 + 60 +12)) * 100 = 10.49 %
% of running time due to the term 12 : (12 / (500 + 60 +12)) * 100 = 2.09 %
The following table shows the growth rate of all the terms of the function
$f(n) = 5n^2 + 6n + 12$

n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41 %	10.49 %	2.09 %
100	98.79 %	1.19 %	0.02 %
1000	99.88 %	0.12 %	0.0002 %
10000	99.99 %	0.01 %	2.4E-06 %

We can see that as n grows, the dominant term n^2 accounts for most of the running time and we can ignore the smaller terms. Calculating exact function f(n) for the time complexity may be difficult. So the terms which do not significantly change the magnitude of function can be dropped from the function. In this way we can get an approximation of the time efficiency and we are satisfied with this approximation because this is very close to the exact value when n becomes large. This approximate measure of complexity is known as asymptotic complexity.

There are some standard functions whose growth rates are known, we find out the complexity of our algorithm and compare it with these known functions whose growth rates are known. The growth rates of some known functions are shown in the table.

	g(n)					
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4.29E+09
64	6	64	384	4096	262144	1.84E+19

Growth rate of standard functions

From the table we see that some functions grow faster than others. The growth rate of $g(n)=\log_2 n$ is least, and the function $g(n)=2^n$ grows very fast. The function $g(n) = n$ grows faster than $\log_2 n$ but slower than $n\log_2 n$ or n^2, n^3, 2^n. To compare the growth rate of function f(n) with these standard functions, we can use big O notation which is described in the next section.

1.5.1 Big O notation

It is the most commonly used notation to measure the performance of any algorithm by defining its order of growth. If f(n) and g(n) are two functions defined for positive integers, then f(n) is O(g(n)) (read as f(n) is big oh of g(n) or f(n) is on the order of g(n)) if there exists constants c and n_0 such that-
$f(n) \leq cg(n)$ for all $n \geq n_0$

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n).

(i) $4n + 3$ is O(n) as there exists constants 5 and 3 such that $4n + 3 \leq 5n$ for all $n \geq 3$.
Here $f(n) = 4n + 3$, $g(n) = n$, $c = 5$ and $n_0 = 3$.

(ii) $5n^2 + 2n + 6$ is $O(n^2)$ as there exists constants 6 and 4 such that $5n^2 + 2n + 6 \leq 6n^2$ for all $n \geq 4$.
Here $f(n)= 5n^2 + 2n + 6$, $g(n) = n^2$, $c = 6$ and $n_0 = 4$.

1.5.1.1 Rules for O notation

(1) Transitivity
If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) is O(h(n))
or we can say that O(O(h(n))) is O(h(n))

(2) If $f_1(n)$ is O(h(n)) and $f_2(n)$ is O(h(n)), then $f_1(n) + f_2(n)$ is O(h(n))
Proof :
$f_1(n)$ is O(h(n)) => there exists c_1 and n_1 such that $f_1(n) \leq c_1 h(n)$ for all $n \geq n_1$
$f_2(n)$ is O(h(n)) => there exists c_2 and n_2 such that $f_2(n) \leq c_2 h(n)$ for all $n \geq n_2$
$f_1(n) + f_2(n) \leq (c_1 + c_2)\, h(n)$ for all $n \geq \max(n_1,n_2)$
if $c_0 = c_1 + c_2$ and $n_0 = \max(n_1,n_2)$
$f_1(n) + f_2(n) \leq c_0\, h(n)$ for all $n \geq n_0$
$f_1(n) + f_2(n)$ is O(h(n))

(3) If $f_1(n)$ is O(h(n)) and $f_2(n)$ is O(g(n)), then $f_1(n) + f_2(n)$ is max(O(h(n)),O(g(n)))
Proof :
$f_1(n)$ is O(h(n)) => there exists c_1 and n_1 such that $f_1(n) \leq c_1 h(n)$ for all $n \geq n_1$
$f_2(n)$ is O(g(n)) => there exists c_2 and n_2 such that $f_2(n) \leq c_2 g(n)$ for all $n \geq n_2$
$f_1(n) + f_2(n) \leq c_1\, h(n) + c_2\, g(n)$ for all $n \geq \max(n_1,n_2)$
if $c = \max(c_1,c_2)$ and $n_0 = \max(n_1,n_2)$
$f_1(n) + f_2(n) \leq c\, h(n) + c\, g(n)$ for all $n \geq n_0$
$f_1(n) + f_2(n) \leq 2c \max(h(n),g(n))$ for all $n \geq n_0$
$f_1(n) + f_2(n) \leq c_0 \max(h(n),g(n))$ for all $n \geq n_0$
$f_1(n) + f_2(n)$ is max(O(h(n)),O(g(n)))

(4) If $f_1(n)$ is O(h(n)) and $f_2(n)$ is O(g(n)) then $f_1(n)f_2(n)$ is O(h(n)g(n))

(5) $f(n) = C$ => $f(n)$ is O(1)
Proof:
$f(n) \leq C * 1$ for any n
So f(n) is O(1)

(6) If $f(n) = C * h(n)$ where C is a constant, then f(n) is O(h(n))
Proof:

f(n) is O(C*h(n))
there exists constants a and b such that
$f(n) \leq a * C * h(n)$ for all $n \geq b$
Taking $c = a*C$ and $n_0 = b$
$f(n) \leq c * h(n)$ for all $n \geq n_0$
So f(n) is O(h(n)).

(7) Any polynomial P(n) of degree m is $O(n^m)$
Proof:
$P(n) = a_0 + a_1 n + a_2 n^2 + \ldots\ldots\ldots + a_m n^m$
Let $c_0 = |a_0|$, $c_1 = |a_1|$, $c_2 = |a_2|$, $\ldots\ldots\ldots$ $c_m = |a_m|$

$P(n) \leq c_0 + c_1 n + c_2 n^2 + \ldots\ldots\ldots + c_m n^m = (c_0/n^m + c_1/n^{m-1} + \ldots\ldots\ldots + c_m)\, n^m$
$\leq (c_0 + c_1 + \ldots\ldots\ldots\ldots + c_m)\, n^m = K.n^m$
$c = K$, and $n_0 = 1$

(8) n^a is $O(n^b)$ only if a<=b , i.e. n^3 is $O(n^3)$ or $O(n^4)$ or $O(n^5)$ but n^4 is not $O(n^3)$

(9) All logarithms grow at the same rate i.e. while computing the O notation, base of the logarithm is not important. To justify this we'll prove that $O(\log_a n)$ is $O(\log_b n)$ and $O(\log_b n)$ is $O(\log_a n)$ for all a, b >1.

$\log_a n$ is $O(\log_b n)$
Let $xa = \log_a n$ and $yb = \log_b n$
$a^{xa} = n$ and $b^{yb} = n$
$a^{xa} = b^{yb}$
Taking $\log_a$ of both sides
$\log_a a^{xa} = \log_a b^{yb}$
$xa * \log_a a = yb * \log_a b$ (since $\log_z m^k = k * \log_z m$ for any z, m and k)
$xa = yb * \log_a b$ (since $\log_a a = 1$)
$\log_a n = \log_b n * \log_a b$ (since $xa = \log_a n$ and $yb = \log_b n$)
$\log_a n = C * \log_b n$ ($C = \log_a b$)
$\log_b n = K * \log_a n$ ($K = 1 / \log_a b$)
We have already seen that if $f(n) = C * h(n)$ where C is a constant, then f(n) is O(h(n)).
So $\log_a n$ is $O(\log_b n)$ and $\log_b n$ is $O(\log_a n)$ for any constants a and b.

(10) log n is O(n)
log n is $O(n^a)$ for any positive constant a
Applying the definition of limits, definition of O notation is equivalent to-

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = c$$

L'Hospital's rule can be used for computing this form of limit and it states that-
If $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} g(n) = \infty$
then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$
where f' and g' represent the derivatives of f and g respectively.

If $f(n) = \log n$ and $g(n) = n$

$$\lim_{n\to\infty} \frac{\log n}{n} = \lim_{n\to\infty} \frac{1/n}{1} = 0$$

So log n is O(n).

If $f(n) = \log n$ and $g(n) = n^a$

$$\lim_{n\to\infty} \frac{\log n}{n^a} = \lim_{n\to\infty} \frac{1/n}{an^{a-1}} = \frac{1}{a\,n^a} = 0 \text{ (for any positive constant a)}$$

So log n is $O(n^a)$ for any positive constant a.

(11) $\log^k n$ is O(n)

Proof:

$\log^k n$ is same as $(\log n)^k$

We have to prove that $(\log n)^k$ is O(n)

For this we have to show that $(\log n)^k \le cn$ for $n \ge n_0$

or we have to show that $\log n \le cn^{1/k}$ for $n \ge n_0$

If we take a =1/k, we have to show that $\log n \le cn^a$ for $n \ge n_0$

or we have to show that log n is $O(n^a)$

We have already proved this.

Now let us see how to calculate the O notation from a given function f(n). Since constants don't matter in the O notation (statement 6), the coefficients of all the terms are set to one. After that we reject all the smaller terms and keep the largest one. The order of terms from smallest to largest is-

O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(n^3)$,.....$O(n^k)$, $O(2^n)$, $O(3^n)$,.....$O(k^n)$, O(n!)

For example consider this function-

$f(n) = \log n + 2n^3 + 5n$

Removing all the coefficients we get-

$\log n + n^3 + n$

Now from O(log n), $O(n^3)$ and O(n), the largest is $O(n^3)$ so we ignore all the smaller terms and say that f(n) is $O(n^3)$.

Here are some more examples-

$f(n) = 23$ is O(1)

$f(n) = 7\log n + 5n + 4$ is O(n)

$f(n) = n^2 + n \log n + n$ is $O(n^2)$

$f(n) = n \log n + \log n$ is O(n log n)

$f(n) = n^2 + n^3 + 2^n$ is $O(2^n)$

$f(n) = 3^n + 5^n + n^7$ is $O(5^n)$

$f(n) = n + 4^n + n!$ is O(n!)

For any given function f(n) there can be many functions such that f(n) is O(g(n)). If there is a function $f(n) = 3n^2 + 5n$, then we say that f(n) is $O(n^2)$, but according to definition it is equally correct to say that f(n) is $O(n^3)$ or $O(n^7)$ or $O(2^n)$ or order of any function that is larger than n^2 in the order list. Although all these are correct statements but they are not informative. The statement f(n) is O(g(n)) will be informative when the function g(n) is the smallest possible function that satisfies f(n) is O(g(n)). There may be many upper bounds on f(n) but we are interested in the least upper bound.

Let us take some common complexities and see the situations in which they occur.

(i) O(1) constant

This is the case when we have to obtain the first element from a set of data, or when we get element in first time itself e.g. hash table. In this case running time is independent of the size of input.

(ii) O(n) linear

This is the case when each element in a set of data has to be processed. Here the running time is linearly proportional to n, i.e. if the size of input is doubled then running time will also be doubled, e.g. worst case of linear search or best case of bubble sort.

(iii) O(log n) logarithmic

This is the case in algorithms where a set of data is repeatedly divided into half and the middle element is processed, e.g. binary search, binary tree traversal.

(iv) O(n log n) linear logarithmic
This is the case when a set of data is repeatedly divided into half and each half is processed independently, e.g. best case of quick sort.

(v) $O(n^2)$ quadratic
This is the case when full set of data has to be traversed for each element of the set or we can say that all pairs of data items are processed e.g. worst case of bubble sort. Quadratic problems are useful only for small sets of data because n^2 increases rapidly as n increases.

(vi) $O(n^3)$ cubic
This is the case when all triplets of data items are processed.

(vi) $O(2^n)$ exponential
This is the case when all possible subsets of a set of data are generated.

(vi) O(n!)
This is the case when all possible permutations of a set of data are generated.

Arrays, Pointers and Structures

2.1 Arrays

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the elements may be any valid data type like char, int or float. The elements of array share the same variable name but each element has a different index number known as subscript.

Consider a situation when we want to store and display the age of 100 employees. We can take an array variable `age` of type int. The size of this array variable is 100 so it is capable of storing 100 integer values. The individual elements of this array are-

`age[0], age[1], age[2], age[3], age[4],.....age[98], age[99]`

In C, the subscripts start from zero, so `age[0]` is the first element, `age[1]` is the second element of array and so on.

Arrays can be single dimensional or multidimensional. The number of subscripts determines the dimension of array. A one-dimensional array has one subscript; two-dimensional array has two subscripts and so on. The two-dimensional arrays are known as matrices.

2.1.1 One Dimensional Array

2.1.1.1 Declaration of 1-D Array

Like other simple variables, arrays should also be declared before they are used in the program. The syntax for declaration of an array is-

```
data_type array_name[size];
```

Here `array_name` denotes the name of the array and it can be any valid C identifier, `data_type` is the data type of the elements of array. The size of the array specifies the number of elements that can be stored in the array. It may be a positive integer constant or constant integer expression. Here are some examples of array declarations-

```
int age[100];
float salary[15];
char grade[20];
```

Here `age` is an integer type array, which can store 100 elements of integer type. The array `salary` is a float type array of size 15, can hold float values and third one is a character type array of size 20, can hold characters. The individual elements of the above arrays are-

```
age[0], age[1], age[2],.....age[99]
salary[0], salary [1], salary [2],.....salary [14]
grade[0], grade[1], grade[2],.....grade[19]
```

When the array is declared, the compiler allocates space in memory sufficient to hold all the elements of the array, so the size of array should be known at the compile time. Hence we can't use variables for specifying the size of array in the declaration. The symbolic constants can be used to specify the size of array. For example-

```
#define SIZE 10
main()
{
        int size = 15;
        float sal[SIZE];    /*Valid*/
        int marks[size];    /*Not valid*/
        ..............
        ..............
}
```

The use of symbolic constant to specify the size of array makes it convenient to modify the program if the size of array is to be changed later, because the size has to be changed only at one place, in the `#define` directive.

2.1.1.2 Accessing 1-D Array Elements

The elements of an array can be accessed by specifying the array name followed by subscript in brackets. In C, the array subscripts start from 0. Hence if there is an array of size 5 then the valid subscripts will be from 0 to 4. The last valid subscript is one less than the size of the array. This last valid subscript is known as the upper bound of the array and 0 is known as the lower bound of the array. Let us take an array-

```
int arr[5];     /*Size of array arr is 5, can hold five integer elements*/
```

The elements of this array are-

```
arr[0], arr[1], arr[2], arr[3], arr[4]
```

Here 0 is the lower bound and 4 is the upper bound of the array.

The subscript can be any expression that yields an integer value. It can be any integer constant, integer variable, integer expression or return value(int) from a function call. For example, if `i` and `j` are integer variables then these are some valid subscripted array elements-

```
arr[3], arr[i], arr[i+j], arr[2*j], arr[i++]
```

A subscripted array element is treated as any other variable in the program. We can store values in them, print their values or perform any operation that is valid for any simple variable of the same data type. For example if `arr` and `sal` are two arrays of sizes 5 and 10 respecively, then these are valid statements-

```
int arr[5];
float sal[10];
int i;
scanf("%d",&arr[1]);   /*input value into arr[1]*/
printf("%f",sal[3]);   /*print value of sal[3]*/
arr[4] = 25;           /*assign a value to arr[4]*/
arr[4]++;              /*Increment the value of arr[4] by 1*/
sal[5]+=200;           /*Add 200 to sal[5]*/
sum = arr[0]+arr[1]+arr[2]+arr[3]+arr[4];/*Add all the values of array arr[5]*/
i=2;
scanf("%f",&sal[i]);   /*Input value into sal[2]*/
printf("%f",sal[i]);   /*Print value of sal[2]*/
printf("%f",sal[i++] ); /*Print value of sal[2] and increment the value of i*/
```

There is no check on bounds of the array. For example, if we have an array `arr` of size 5, the valid subscripts are only 0, 1, 2, 3, 4 and if someone tries to access elements beyond these subscripts, like `arr[5]` , `arr[10]`, the compiler will not show any error message but this may lead to run time errors. So it is the responsibility of programmer to provide array bounds checking wherever needed.

2.1.1.3 Processing 1-D Arrays

For processing arrays we generally use a for loop and the loop variable is used at the place of subscript. The initial value of loop variable is taken 0 since array subscripts start from zero. The loop variable is increased by 1 each time so that we can access and process the next element in the array. The total number of passes in the

loop will be equal to the number of elements in the array and in each pass we will process one element. Suppose arr is an array of type int-

(i) Reading values in arr
```
for(i=0; i<10; i++)
        scanf("%d",&arr[i]);
```
(ii) Displaying values of arr
```
for(i=0; i<10; i++)
     printf("%d ",arr[i]);
```
(iii) Adding all the elements of arr
```
sum=0;
for(i=0; i<10; i++)
        sum+=arr[i];
```

```
/*P2.1 Program to input values into an array and display them*/
#include<stdio.h>
main()
{
        int arr[5], i;
        for(i=0; i<5; i++)
        {
                printf("Enter the value for arr[%d] : ",i);
                scanf("%d",&arr[i]);
        }
        printf("The array elements are : \n");
        for(i=0; i<5; i++)
                printf("%d\t",arr[i]);
        printf("\n");
}
```

2.1.1.4 Initialization of 1-D Array

After declaration, the elements of a local array have garbage value while the elements of global and static arrays are automatically initialized to zero. We can explicitly initialize arrays at the time of declaration. The syntax for initialization of an array is-

```
data_type array_name[size]={value1, value2.....valueN};
```

Here array_name is the name of the array variable, size is the size of the array and value1, value2, ...valueN are the constant values known as initializers, which are assigned to the array elements one after another. These values are separated by commas and there is a semicolon after the ending braces. For example-

```
int marks[5] = {50, 85, 70, 65, 95};
```

The values of the array elements after this initialization are-

```
marks[0]:50, marks[1]:85, marks[2]:70, marks[3]:65, marks[4]:95
```

While initializing a 1-D array, it is optional to specify the size of the array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers. For example-

```
int marks[] = {99, 78, 50, 45, 67, 89};
float sal[] = {25.5, 38.5, 24.7};
```

Here the size of array marks is assumed to be 6 and that of sal is assumed to be 3.

If during initialization the number of initializers is less than the size of array, then all the remaining elements of array are assigned value zero. For example-

```
int marks[5] = {99, 78};
```

Here the size of array is 5 while there are only 2 initializers. After this initialization the value of the elements are-

```
marks[0]:99,  marks[1]:78,  marks[2]:0,  marks[3]:0,  marks[4]:0
```

So if we initialize an array like this-

```
int arr[100] = {0};
```

then all the elements of `arr` will be initialized to zero.

If the number of initializers is more than the size given in brackets then compiler will show an error. For example-

```
int arr[5] = {1,2,3,4,5,6,7,8};        /*Error*/
```

We can't copy all the elements of an array to another array by simply assigning it to the other array. For example if we have two arrays `a` and `b`, then-

```
int a[5] = {1,2,3,4,5};
int b[5];
b = a;  /*Not valid*/
```

We will have to copy all the elements of array one by one, using a for loop.

```
for(i=0; i<5; i++)
        b[i] = a[i];
```

In the following program we will find out the largest and smallest number in an integer array.

```
/*P2.2 Program to find the largest and smallest number in an array*/
#include<stdio.h>
main()
{
        int i,arr[10]={2,5,4,1,8,9,11,6,3,7};
        int small,large;
        small = large = arr[0];
        for(i=1; i<10; i++)
        {
                if(arr[i] < small)
                        small = arr[i];
                if(arr[i] > large)
                        large = arr[i];
        }
        printf("Smallest = %d, Largest = %d\n",small,large);
}
```

We have taken the value of first element as the initial value of `small` and `large`. Inside the for loop, we will start comparing from second element onwards so this time we have started the loop from 1 instead of 0.

The following program will reverse the elements of an array.

```
/*P2.3 Program to reverse the elements of an array*/
#include<stdio.h>
main()
{
        int i,j,temp,arr[10] = {1,2,3,4,5,6,7,8,9,10};
        for(i=0,j=9; i<j; i++,j--)
        {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
        }
        printf("After reversing, the array is : ");
        for(i=0; i<10; i++)
                printf("%d  ",arr[i]);
        printf("\n");
}
```

In the `for` loop we have used comma operator and taken two variables `i` and `j`. The variable `i` is initialized with the lower bound and `j` is initialized with upper bound. After each pass of the loop, `i` is incremented while `j` is decremented. Inside the loop, `a[i]` is exchanged with `a[j]`. So `a[0]` will be exchanged with `a[9]`, `a[1]` with `a[8]`, `a[2]` with `a[7]` and so on.

2.1.1.5 1-D Arrays and Functions

2.1.1.5.1 Passing Individual Array Elements to a Function

We know that an array element is treated as any other simple variable in the program. So like other simple variables, we can pass individual array elements as arguments to a function.

```
/*P2.4 Program to pass array elements to a function*/
#include<stdio.h>
void check(int num);
main()
{
        int arr[10],i;
        printf("Enter the array elements : ");
        for(i=0; i<10; i++)
        {
                scanf("%d",&arr[i]);
                check(arr[i]);
        }
}
void check(int num)
{
        if(num%2 == 0)
                printf("%d is even\n",num);
        else
        printf("%d is odd\n",num);
}
```

2.1.1.5.2 Passing whole 1-D Array to a Function

We can pass whole array as an actual argument to a function. The corresponding formal argument should be declared as an array variable of the same data type.

```
main()
{

        int arr[10]
        .....
        .....
        func(arr); /*In function call, array name is specified without brackets*/
}
func(int val[10])
{
        .....
        .....
}
```

It is optional to specify the size of the array in the formal argument, for example we may write the function definition as-

```
func(int val[])
{
  .....
  .....
}
```

The mechanism of passing an array to a function is quite different from that of passing a simple variable. In the case of simple variables, the called function creates a copy of the variable and works on it, so any changes made in the function do not affect the original variable. When an array is passed as an actual argument, the called function actually gets access to the original array and works on it, so any changes made inside the function affect the original array. Here is a program in which an array is passed to a function.

```
/*P2.5 Program to pass an array to a function*/
#include<stdio.h>
```

```
void func(int val[]);
main()
{
        int i,arr[6] = {1,2,3,4,5,6};
        func(arr);
        printf("Contents of array are now : ");
        for(i=0; i<6; i++)
                printf("%d ",arr[i]);
        printf("\n");
}
void func(int val[])
{
        int sum=0,i;
        for(i=0;i<6;i++)
        {
                val[i] = val[i]*val[i];
                sum += val[i];
        }
        printf("The sum of squares = %d\n",sum);
}
```

Output:
The sum of squares = 91
Contents of array are now : 1 4 9 16 25 36

Here we can see that the changes made to the array inside the called function are reflected in the calling function. The name of the formal argument is different but it refers to the original array.

2.1.2 Two Dimensional Arrays

2.1.2.1 Declaration and Accessing Individual Elements of a 2-D array

The syntax of declaration of a 2-D array is similar to that of 1-D arrays, but here we have two subscripts.

```
data_type array_name[rowsize][columnsize];
```

Here rowsize specifies the number of rows and columnsize represents the number of columns in the array. The total number of elements in the array are rowsize * columnsize. For example-

```
int arr[4][5];
```

Here `arr` is a 2-D array with 4 rows and 5 columns. The individual elements of this array can be accessed by applying two subscripts, where the first subscript denotes the row number and the second subscript denotes the column number. The starting element of this array is `arr[0][0]` and the last element is `arr[3][4]`. The total number of elements in this array is 4*5 = 20.

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
Row 1	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
Row 2	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]
Row 3	arr[3][0]	arr[3][1]	arr[3][2]	arr[3][3]	arr[3][4]

2.1.2.2 Processing 2-D Arrays

For processing 2-D arrays, we use two nested for loops. The outer `for` loop corresponds to the row and the inner `for` loop corresponds to the column.

```
int arr[4][5];
```

(i) Reading values in `arr`

```
        for(i=0; i<4; i++)
                for(j=0; j<5; j++)
                        scanf("%d",&arr[i][j]);
```

(ii) Displaying values of `arr`

```
        for(i=0; i<4; i++)
                for(j=0; j<5; j++)
                        printf("%d ",arr[i][j]);
```

This will print all the elements in the same line. If we want to print the elements of different rows on different lines then we can write like this-

```
for(i=0; i<4; i++)
{
        for(j=0; j<5; j++)
                printf("%d ",arr[i][j]);
        printf("\n");
}
```

Here the `printf("\n")` statement causes the next row to begin from a new line.

```
/*P2.6 Program to input and display a matrix*/
#define ROW 3
#define COL 4
#include<stdio.h>
main()
{
        int mat[ROW][COL], i, j;
        printf("Enter the elements of the matrix(%dx%d) row-wise :\n",ROW,COL);
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        scanf("%d",&mat[i][j]);

        printf("The matrix that you have entered is :\n");
        for(i=0; i<ROW; i++)
        {
                for(j=0; j<COL; j++)
                        printf("%5d",mat[i][j]);
                printf("\n");
        }
        printf("\n");
}
```

2.1.2.3 Initialization of 2-D Arrays

2-D arrays can be initialized in a way similar to that of 1-D arrays. For example-

```
int mat[4][3] = {11,12,13,14,15,16,17,18,19,20,21,22};
```

These values are assigned to the elements row-wise, so the values of elements after this initialization are-

mat[0][0] : 11	mat[0][1] : 12	mat[0][2] : 13
mat[1][0] : 14	mat[1][1] : 15	mat[1][2] : 16
mat[2][0] : 17	mat[2][1] : 18	mat[2][2] : 19
mat[3][0] : 20	mat[3][1] : 21	mat[3][2] : 22

While initializing we can group the elements row-wise using inner braces. For example-

```
int mat[4][3] = { {11,12,13}, {14,15,16}, {17,18,19}, {20,21,21} };
int mat[4][3] = {
                        {11,12,13},     /*Row 0*/
                        {14,15,16},     /*Row 1*/
                        {17,18,19},     /*Row 2*/
                        {20,21,21}      /*Row 3*/
                };
```

Here the values in the first inner braces will be the values of Row 0, values in the second inner braces will be values of Row 1 and so on. Now consider this array initialization-

```
int mat[4][3] = {
                        {11},           /*Row 0*/
                        {12,13},        /*Row 1*/
                        {14,15,16},     /*Row 2*/
                        {17}            /*Row 3*/
                };
```

The remaining elements in each row will be assigned values 0, so the values of elements will be-

mat[0][0] : 11	mat[0][1] : 0	mat[0][2] : 0
mat[1][0] : 12	mat[1][1] : 13	mat[1][2] : 0
mat[2][0] : 14	mat[2][1] : 15	mat[2][2] : 16
mat[3][0] : 17	mat[3][1] : 0	mat[3][2] : 0

In 2-D arrays, it is optional to specify the first dimension while initializing but the second dimension should always be present. For example-

```
int mat[][3] = {
                        {1,10},
                        {2,20,200},
                        {3},
                        {4,40,400}
                };
```

Here first dimension is taken 4 since there are 4 rows in initialization list.

A 2-D array is also known as a matrix. The next program adds two matrices; the order of both the matrices should be same.

```
/*P2.7 Program for addition of two matrices.*/
#define ROW 3
#define COL 4
#include<stdio.h>
main()
{
        int i,j,mat1[ROW][COL],mat2[ROW][COL],mat3[ROW][COL];
        printf("Enter matrix mat1(%dx%d)row-wise :\n",ROW,COL);
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        scanf("%d",&mat1[i][j]);
        printf("Enter matrix mat2(%dx%d)row-wise :\n",ROW,COL);
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        scanf("%d",&mat2[i][j] );
        /*Addition*/
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        mat3[i][j] = mat1[i][j] + mat2[i][j];
        printf("The resultant matrix mat3 is :\n");
        for(i=0; i<ROW; i++)
        {
                for(j=0; j<COL; j++)
                        printf("%5d",mat3[i][j]);
                printf("\n");
        }
}
```

Now we will write a program to multiply two matrices. Multiplication of matrices requires that the number of columns in first matrix should be equal to the number of rows in second matrix. Each row of first matrix is multiplied with the column of second matrix then added to get the element of resultant matrix. If we multiply two matrices of order m x n and n x p then the multiplied matrix will be of order m x p. For example-

$$A_{2x2} = \begin{bmatrix} 4 & 5 \\ 3 & 2 \end{bmatrix} \qquad B_{2x3} = \begin{bmatrix} 2 & 6 & 3 \\ -3 & 2 & 4 \end{bmatrix}$$

$$C_{2x3} = \begin{bmatrix} 4*2+5*(-3) & 4*6+5*2 & 4*3+5*4 \\ 3*2+2*(-3) & 3*6+2*2 & 3*3+2*4 \end{bmatrix} = \begin{bmatrix} -7 & 34 & 32 \\ 0 & 22 & 17 \end{bmatrix}$$

```
/*P2.8 Program for multiplication of two matrices*/
#include<stdio.h>
#define ROW1 3
```

```
#define COL1 4
#define ROW2 COL1
#define COL2 2
main()
{
        int mat1[ROW1][COL1],mat2[ROW2][COL2],mat3[ROW1][COL2];
        int i,j,k;
        printf("Enter matrix mat1(%dx%d)row-wise :\n",ROW1,COL1);
        for(i=0; i<ROW1; i++)
                for(j=0; j<COL1; j++)
                        scanf("%d",&mat1[i][j]);
        printf("Enter matrix mat2(%dx%d)row-wise :\n",ROW2,COL2);
        for(i=0; i<ROW2; i++)
                for(j=0; j<COL2; j++)
                        scanf("%d",&mat2[i][j] );
        /*Multiplication*/
        for(i=0; i<ROW1; i++)
                for(j=0; j<COL2; j++)
                {
                        mat3[i][j] = 0;
                        for(k=0; k<COL1; k++)
                                mat3[i][j] += mat1[i][k] * mat2[k][j];
                }
        printf("The Resultant matrix mat3 is :\n");
        for(i=0; i<ROW1; i++)
        {
                for(j=0; j<COL2; j++)
                        printf("%5d",mat3[i][j]);
                printf("\n");
        }
}
```

2.2 Pointers

A pointer is a variable that stores memory address. Like all other variables it also has a name, has to be declared and occupies some space in memory. It is called pointer because it points to a particular location in memory by storing the address of that location. The use of pointers makes the code more efficient and compact. Some of the uses of pointers are-

(i) Accessing array elements.
(ii) Returning more than one value from a function.
(iii) Accessing dynamically allocated memory
(iv) Implementing data structures like linked lists, trees, and graphs.

2.2.1 Declaration of a Pointer Variable

Like other variables, pointer variables should also be declared before being used. The general syntax of declaration is-

```
data_type *pname;
```

Here `pname` is the name of the pointer variable, which should be a valid C identifier. The asterisk '*' preceding this name informs the compiler that the variable is declared as a pointer. Here `data_type` is known as the base type of pointer. Let us take some pointer declarations-

```
int *iptr;
float *fptr;
char *cptr,ch1,ch2;
```

Here `iptr` is a pointer that should point to variable of type `int`, similarly `fptr` and `cptr` should point to variables of `float` and `char` type respectively. Here type of variable `iptr` is 'pointer to int' or (int *), or we can say that base type of `iptr` is `int`. We can also combine the declaration of simple variables and pointer

variables as we have done in the third declaration statement where ch1 and ch2 are declared as variables of type char.

Pointers are also variables so compiler will reserve space for them and they will also have some address. All pointers irrespective of their base type will occupy same space in memory since all of them contain addresses only. The size of a pointer depends on the architecture and may vary on different machines; in our discussion we will take the size of pointer to be 4 bytes.

2.2.2 Assigning Address to a Pointer Variable

When we declare a pointer variable it contains garbage value i.e. it may be pointing anywhere in the memory. So we should always assign an address before using it in the program. The use of an unassigned pointer may give unpredictable results and even cause the program to crash. Pointers may be assigned the address of a variable using assignment statement. For example-

```
int *iptr,age = 30;
float *fptr,sal = 1500.50;
iptr = &age;
fptr = &sal;
```

Now iptr contains the address of variable age i.e. it points to variable age, similarly fptr points to variable sal. Since iptr is declared as a pointer of type int, we should assign address of only integer variables to it. If we assign address of some other data type then compiler won't show any error but the output will be incorrect.

We can also initialize the pointer at the time of declaration, but in this case the variable should be declared before the pointer: For example-

```
int age=30,*iptr=&age;
float sal=1500.50,*fptr=&sal;
```

It is also possible to assign the value of one pointer variable to the other, provided their base type is same. For example if we have an integer pointer p1 then we can assign the value of iptr to it as-

```
p1 = iptr;
```

Now both pointer variables iptr and p1 contain the address of variable age and point to the same variable age.

We can assign constant zero to a pointer of any type. A symbolic constant NULL is defined in the header file stdio.h, which denotes the value zero. The assignment of NULL to a pointer guarantees that it does not point to any valid memory location. This can be done as-

```
ptr = NULL;
```

2.2.3 Dereferencing Pointer Variables

We can access a variable indirectly using pointers. For this we will use the indirection operator(*). By placing the indirection operator before a pointer variable, we can access the variable whose address is stored in the pointer. Let us take an example-

```
int a = 87
float b = 4.5;
int *p1 = &a;
float *p2 = &b;
```

In our program, if we place '*' before p1 then we can access the variable whose address is stored in p1. Since p1 contains the address of variable a, we can access the variable a by writing *p1. Similarly we can access variable b by writing *p2. So we can use *p1 and *p2 in place of variable names a and b anywhere in our program. Let us see some examples-

*p1= 9;	is equivalent to	a = 9;
(*p1)++;	is equivalent to	a++;
x = *p2 + 10;	is equivalent to	x = b + 10;

```
printf("%d %f",*p1,*p2);    is equivalent to    printf("%d %f",a,b);
scanf("%d%f",p1,p2);        is equivalent to    scanf("%d%f",&a,&b);
```

The indirection operator can be read as 'value at the address'. For example *p1 can be read as 'value at the address p1'. This indirection operator (*) is different from the asterisk that was used while declaring the pointer variable.

```
/*P2.9 Program to dereference pointer variables*/
#include<stdio.h>
main()
{
        int a = 87;
        float b = 4.5;
        int *p1 = &a;
        float *p2 = &b;
        printf("Value of p1 = Address of a = %p\n",p1);
        printf("Value of p2 = Address of b = %p\n",p2);
        printf("Address of p1 = %p\n",&p1);
        printf("Address of p2 = %p\n",&p2);
        printf("Value of a = %d %d %d\n",a,*p1,*(&a));
        printf("Value of b = %f %f %f\n",b,*p2,*(&b));
}
```

2.2.4 Pointer to Pointer

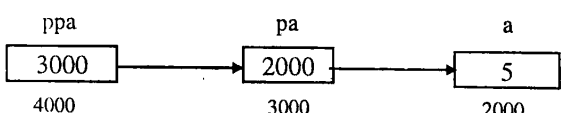
We know that pointer is a variable that can contain memory address. The pointer variable takes some space in memory and therefore it also has an address. We can store the address of a pointer variable in some other variable, which is known as a pointer to pointer variable. Similarly we can have a pointer to pointer to pointer variable and this concept can be extended to any limit, but in practice only pointer to pointer is used. Pointer to pointer is generally used while passing pointer variables to functions. The syntax of declaring a pointer to pointer is as-

```
data_type **pptr;
```

Here variable pptr is a pointer to pointer and it can point to a pointer pointing to a variable of type data_type. The double asterisk used in the declaration informs the compiler that a pointer to pointer is being declared. Now let us take an example-

```
int a = 5;
int *pa = &a;
int **ppa = &pa;
```

Here type of variable a is int, type of variable pa is (int *) or pointer to int, and type of variable ppa is (int **) or pointer to pointer to int.



Here pa is a pointer variable, which contains the address of the variable a and ppa is a pointer to pointer variable, which contains the address of the pointer variable pa.

We know that *pa gives value of a, similarly *ppa will give the value of pa. Now let us see what value will be given by **ppa.

```
    **ppa
→ *(*ppa)
→ *pa  (Since *ppa gives pa)
→ a    (Since *pa gives a)
```

Hence we can see that **ppa will give the value of a. So to access the value indirectly pointed to by a pointer to pointer, we can use double indirection operator. The table given next will make this concept clear.

Value of a	a	*pa	**ppa	5
Address of a	&a	pa	*ppa	2000
Value of pa	&a	pa	*ppa	2000
Address of pa		&pa	ppa	3000
Value of ppa		&pa	ppa	3000
Address of ppa			&ppa	4000

```
/*P2.10 Program to understand pointer to pointer*/
#include<stdio.h>
main()
{
        int a = 5;
        int *pa;
        int **ppa;
        pa = &a;
        ppa = &pa;
        printf("Address of a = %p\n",&a);
        printf("Value of pa = Address of a = %p\n",pa);
        printf("Value of *pa = Value of a = %d\n",*pa);
        printf("Address of pa = %p\n",&pa);
        printf("Value of ppa = Address of pa = %p\n",ppa);
        printf("Value of *ppa = Value of  pa = %p\n",*ppa);
        printf("Value of **ppa = Value of a = %d\n",**ppa);
        printf("Address of ppa = %p\n",&ppa);
}
```

2.2.5 Pointers and One Dimensional Arrays

The elements of an array are stored in contiguous memory locations. Suppose we have an array `arr` of type int.

```
int arr[5] = {1,2,3,4,5};
```

Suppose it is stored in memory at address 5000

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
1	2	3	4	5
5000	5004	5008	5012	5016

Here 5000 is the address of first element, and since each element (type `int`) takes 4 bytes, address of next element is 5004, and so on. The address of first element of the array is also known as the base address of the array. Thus it is clear that the elements of an array are stored sequentially in memory one after another.

In C language, pointers and arrays are closely related. We can access the array elements using pointer expressions. Actually the compiler also accesses the array elements by converting subscript notation to pointer notation. Following are the main points for understanding the relationship of pointers with arrays.

1. Elements of an array are stored in consecutive memory locations.
2. The name of an array is a constant pointer that points to the first element of the array, i.e. it stores the address of the first element, also known as the base address of array.
3. According to pointer arithmetic, when a pointer variable is incremented, it points to the next location of its base type.

For example-

```
int arr[5] = {5,10,15,20,25}
```

Here `arr` is an array that has 5 elements each of type `int`.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
5	10	15	20	25
2000	2004	2008	2012	2016

We can get the address of an element of array by applying & operator in front of subscripted variable name. Hence `&arr[0]` gives address of 0^{th} element, `&arr[1]` gives the address of first element and so on. Since array subscripts start from 0, we'll refer to the first element of array as 0^{th} element and so on. The following program shows that the elements of an array are stored in consecutive memory locations.

```
/*P2.11 Program to print the value and address of the elements of an array*/
#include<stdio.h>
main()
{
        int arr[5] = {5,10,15,20,25};
        int i;
        for(i=0; i<5; i++)
        {
                printf("Value of arr[%d] = %d\t",i,arr[i]);
                printf("Address of arr[%d] = %p\n",i,&arr[i]);
        }
}
```

The name of the array 'arr' denotes the address of 0^{th} element of array which is 2000. The address of 0^{th} element can also be given by `&arr[0]`, so `arr` and `&arr[0]` represent the same address. The name of an array is a constant pointer, and according to pointer arithmetic when an integer is added to a pointer then we get the address of next element of same base type. Hence `(arr+1)` will denote the address of the next element `arr[1]`. Similarly `(arr+2)` denotes the address of `arr[2]` and so on. In other words we can say that the pointer expression `(arr+1)` points to 1^{st} element of array, `(arr+2)` points to 2^{nd} element of array and so on.

arr	→	Points to 0^{th} element	→	&arr[0]	→	2000
arr+1	→	Points to 1^{st} element	→	&arr[1]	→	2004
arr+2	→	Points to 2^{nd} element	→	&arr[2]	→	2008
arr+3	→	Points to 3^{rd} element	→	&arr[3]	→	2012
arr+4	→	Points to 4^{th} element	→	&arr[4]	→	2016

In general we can write-
The pointer expression `(arr+i)` denotes the same address as `&arr[i]`.

Now if we dereference `arr`, then we get the 0^{th} element of array. i.e. expression `*arr` or `*(arr+0)` represents 0^{th} element of array. Similarly on derferencing `(arr+1)` we get the 1^{st} element and so on.

*arr	→	Value of 0^{th} element	→	arr[0]	→	5
*(arr+1)	→	Value of 1^{st} element	→	arr[1]	→	10
*(arr+2)	→	Value of 2^{nd} element	→	arr[2]	→	15
*(arr+3)	→	Value of 3^{rd} element	→	arr[3]	→	20
*(arr+4)	→	Value of 4^{th} element	→	arr[4]	→	25

In general we can write-
`*(arr+i)` → `arr[i]`

So in subscript notation the address of an array element is `&arr[i]` and its value is `arr[i]`, while in pointer notation the address is `(arr+i)` and the element is `*(arr+i)`.

```
/*P2.12 Program to print the value and address of elements of an array using pointer
notation*/
#include<stdio.h>
main()
{
        int i, arr[5] = {5,10,15,20,25};
        for(i=0; i<5; i++)
        {
                printf("Value of arr[%d] = %d\t",i,*(arr+i));
                printf("Address of arr[%d] = %p\n",i,arr+i);
        }
}
```

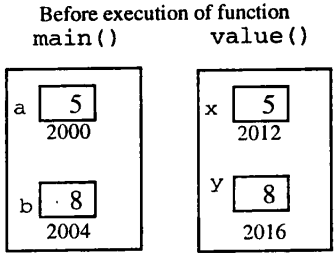
2.2.6 Pointers and Functions

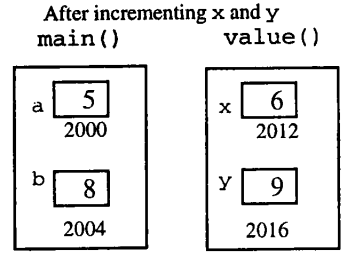
The arguments to the functions can be passed in two ways-
(i) Call by value (ii) Call by reference
In call by value, only the values of arguments are sent to the function while in call by reference, addresses of arguments are sent to the function. In call by value method, any changes made to the formal arguments do not change the actual argument. In call by reference method, any changes made to the formal arguments changes the actual arguments also. C uses only call by value when passing arguments to a function, but we can simulate call by reference by using pointers. All the functions that we had written so far used call by value method. Here is another simple program that uses call by value-

```
/*P2.13 Call by value*/
#include<stdio.h>
void value(int x,int y);
main()
{
        int a=5,b=8;
        printf("Before calling the function,a = %d and b = %d\n",a,b);
        value(a,b);
        printf("After calling the function,a = %d and b = %d\n",a,b);
}
void value(int x,int y)
{
        x++;
        y++;
        printf("Inside function x = %d,y = %d\n",x,y);
}
```

Here a and b are variables declared in the function `main()` while x and y are declared in the function `value()`.These variables reside at different addresses in memory. Whenever the function `value()` is called, two variables are created named x and y and are initialized with the values of variables a and b. This type of parameter passing is called call by value since we are only supplying the values of actual arguments to the calling function. Any operation performed on variables x and y in the function `value()`, will not affect variables a and b.

Before calling the function `value()`, the value of a is 5 and value of b is 8. The values of a and b are copied into x and y. Since the memory locations of x, y and a, b are different, when the values of x and y are incremented, there will be no affect on the values of a and b. Therefore after calling the function, a and b are same as before calling the function and have the values 5 and 8.





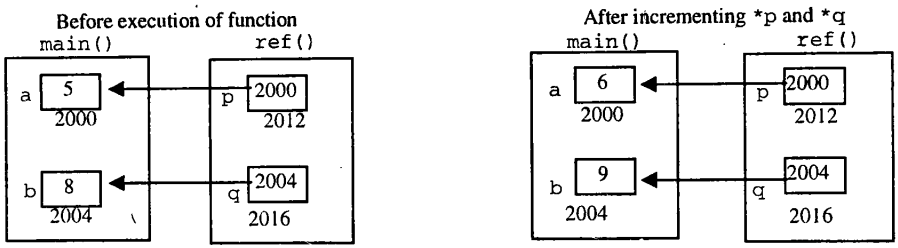
Although C does not use call by reference, we can simulate it by passing addresses of variables as arguments to the function. To accept the addresses inside the function, we will need pointer variables. Here is a program that simulates call by reference by passing addresses of variables a and b.

```
/*P2.14 Call by reference*/
#include<stdio.h>
void ref(int *p,int *q);
main()
{
        int a = 5;
```

```
        int b = 8;
        printf("Before calling the function,a = %d and b = %d\n",a,b);
        ref(&a,&b);
        printf("After calling the function,a = %d and b = %d\n",a,b);
}
void ref(int *p,int *q)
{
        (*p)++;
        (*q)++;
        printf("Inside function *p = %d,*q = %d\n",*p,*q);
}
```

Here we are passing addresses of variables a and b in the function call. So the receiving formal arguments in the function declaration should be declared of pointer type. Whenever function ref() is called, two pointer to int variables, named p and q are created and they are initialized with the addresses of a and b. Now if we dereference pointers p and q, we will be able to access variables a and b from function ref().

The main() accesses the memory locations occupied by variables a and b by writing their names, while ref() accesses the same memory locations indirectly by writing *p, *q.



Before calling the function ref(), the value of a is 5 and value of b is 8. The value of actual arguments are copied into pointer variables p and q, and here the actual arguments are addresses of variables a and b. Since p contains address of variable a, we can access variable a inside ref() by writing *p, similarly variable b can be accessed by writing *q.

Now (*p)++ means value at address 2000 (which is 5) is incremented. Similarly (*q)++ means value at address 2004 (which is 8) is incremented. Now the value of *p = 6 and *q = 9. When we come back to main(), we see that the values of variable a and b have changed. This is because the function ref() knew the addresses of a and b, so it was able to access them indirectly.

So in this way we could simulate call by reference by passing addresses of arguments. This method is mainly useful when the called function has to return more than one values to the function.

2.2.7 Returning More Than One Value from a Function

We have studied that we can return only one value from a function through return statement. This limitation can be overcome by using call by reference. Let us take a simple example to understand this concept. Suppose we want a function to return the sum, difference and product of two numbers passed to it. If we use return statement then we will have to make three different functions with one return statement in each. The following program shows how we can return all these values from a single function.

```
/*P2.15 Program to show how to return more than one value from a function using call by
reference*/
#include<stdio.h>
func(int x,int y,int *ps,int *pd,int *pp);
main()
{
        int a,b,sum,diff,prod;
        a = 6;
        b = 4;
        func(a,b,&sum,&diff,&prod);
        printf("Sum = %d, Difference = %d, Product = %d\n",sum,diff,prod);
```

```
}
func(int x,int y,int *ps,int *pd,int *pp)
{
        *ps = x+y;
        *pd = x-y;
        *pp = x*y;
}
```

In `func()`, variables `a` and `b` are passed by value while variables `sum`, `diff`, `prod` are passsed by reference. The function `func()` gets the addresses of variables `sum`, `diff` and `prod`, so it accesses these variables indirectly using pointers and changes their values.

2.2.8 Function Returning Pointer

We can have a function that returns a pointer. The syntax of declaration of such type of function is -

```
type *func(type1,type 2, ...);
```

For example-

```
float *func1(int,char);  /*This function returns a pointer to float*/
int *func2(int,int);     /*This function returns a pointer to int*/
```

While returning a pointer, make sure that the memory address returned by the pointer will exist even after the termination of function. For example a function of this form should not be written.

```
main()
{
        int *ptr;
        ptr = func();
        .....
}
int *func()
{
        int x = 5;
        int *p = &x;
        .....
        return p;
}
```

Here we are returning a pointer which points to a local variable. We know that a local variable exists only inside the function. Suppose the variable `x` is stored at address 2500, the value of `p` will be 2500 and this value will be returned by the function `func()`. As soon as `func()` terminates, the local variable `x` will cease to exist.

The address returned by `func()` is assigned to pointer variable `ptr` inside `main()`, so now `ptr` will contain address 2500. When we dereference `ptr`, we are trying to access the value of a variable that no longer exists. So never return a pointer which points to a local variable. Now let us take another program that uses a function returning pointer.

```
/*P2.16 Program to show a function that returns pointer*/
#include<stdio.h>
int *fun(int *p, int n);
main()
{
        int arr[10] = {1,2,3,4,5,6,7,8,9,10},n,*ptr;
        n = 5;
        ptr = fun(arr,n);
        printf("arr = %p,ptr = %p,*ptr = %d\n",arr,ptr,*ptr);
}
int *fun(int *p,int n)
{
        p = p+n;
        return p;
}
```

2.2.9 Passing a 1-D Array to a Function

When an array is passed to a function, the changes made inside the function affect the original array. This is because the function gets access to the original array. The following program shows this-

```
/*P2.17 Program to show that changes to the array made inside the function affect the
original array*/
#include<stdio.h>
void func(int a[]);
main()
{
        int i, arr[5] = {3,6,2;7,1};
        func(arr);
        printf("Inside main() : ");
        for(i=0; i<5; i++)
              printf("%d  ",arr[i]);
        printf("\n");
}
void func(int a[])
{
        int i;
        printf("Inside func() : ");
        for(i=0; i<5; i++)
        {
               a[i] = a[i] + 2;
               printf("%d  ",a[i]);
        }
        printf("\n");
}
```

Now let us see what actually happens when an array is passed to a function. There are three ways of declaring the formal parameter, which has to receive the array. We can declare it as an unsized or sized array or we can declare it as a pointer.

```
func(int a[])
{
        .....
}
func(int a[5]);
{
        .....
}
func(int *a);
{
        .....
}
```

In all the three cases the compiler reserves space only for a pointer variable inside the function. In the function call, the array name is passed without any subscript or address operator. Since array name represents the address of first element of array, this address is assigned to the pointer variable in the function. So inside the function we have a pointer that contains the base address of the array. In the above program, the argument `a` is declared as a pointer variable whose base type is `int`, and it is initialized with the base address of array `arr`. We have studied that if we have a pointer variable containing the base address of an array, then we can access any array element either by pointer notation or subscript notation. So inside the function, we can access any i^{th} element of arr by writing `*(a+i)` or `a[i]`. Since we are directly accessing the original array, all the changes made to the array in the called function are reflected in the calling function. The following program will illustrate the point that we have discussed.

```
/*P2.18 When an array is passed to a function, the receiving argument is declared as a
pointer*/
#include<stdio.h>
func(float f[],int *i,char c[5]);
main()
```

```
{
        float f_arr[5] = {1.4,2.5,3.7,4.1,5.9};
        int i_arr[5] = {1,2,3,4,5};
        char c_arr[5] = {'a','b','c','d','e'};
        printf("Inside main(): ");
        printf("Size of f_arr = %u\t",sizeof(f_arr));
        printf("Size of i_arr = %u\t",sizeof(i_arr));
        printf("Size of c_arr = %u\n",sizeof(c_arr));
        func(f_arr,i_arr,c_arr);
}
func(float f[],int *i,char c[5])
{
        printf("Inside func(): ");
        printf("Size of f = %u\t",sizeof(f));
        printf("Size of i = %u\t",sizeof(i));
        printf("Size of c = %u\n",sizeof(c));
}
```

Inside the function `func()`, variables `f`, `i` and `c` are declared as pointers.

2.2.10 Array of Pointers

We can declare an array that contains pointers as its elements. Every element of this array is a pointer variable that can hold address of any variable of appropriate type. The syntax of declaring an array of pointers is similar to that of declaring arrays except that an asterisk is placed before the array name.

```
datatype *arrayname[size];
```

An array of size 10 containing integer pointers can be declared as-

```
int *arrp[10];
```

```
/*P2.19 Array of pointers*/
#include<stdio.h>
main()
{
        int *pa[3];
        int i,a=5,b=10,c=15;
        pa[0] = &a;
        pa[1] = &b;
        pa[2] = &c;
        for(i=0; i<3; i++)
        {
                printf("pa[%d] = %p\t",i,pa[i]);
                printf("*pa[%d] = %d\n",i,*pa[i]);
        }
}
```

Here `pa` is declared as an array of pointers. Every element of this array is a pointer to an integer. `pa[i]` gives the value of the i^{th} element of `pa` which is an address of any `int` variable and `*pa[i]` gives the value of that `int` variable. The array of pointers can also contain addresses of elements of another array.

2.3 Dynamic Memory Allocation

The memory allocation that we have done till now was static memory allocation. The memory that could be used by the program was fixed i.e. we could not increase or decrease the size of memory during the execution of a program. In many applications it is not possible to predict how much memory would be needed by the program at run time. Suppose we declare an array of integers-

```
int emp_no[200];
```

In an array, it is must to specify the size of array while declaring, so the size of this array will be fixed during runtime. Now two types of problems may occur. The first type of problem is that the number of values to be stored is less than the size of array and hence there is wastage of memory. For example if we have to store only

50 values in the above array, then space for 150 values(600 bytes) is wasted. The second type of problem is that, our program fails if we want to store more values than the size of array, for example when there is need to store 205 values in the above array.

To overcome these problems we should be able to allocate memory at run time. The process of allocating memory at the time of execution is called dynamic memory allocation. The allocation and release of this memory space can be done with the help of some built-in-functions whose prototypes are found in and stdlib.h header file. These functions allocate memory from a memory area called heap and release this memory whenever not required, so that it can be used again for some other purpose.

Pointers play an important role in dynamic memory allocation because we can access the dynamically allocated memory only through pointers.

2.3.1 malloc( )

Declaration : `void *malloc(size_t size);`

This function is used to allocate memory dynamically. The argument `size` specifies the number of bytes to be allocated. The type `size_t` is defined in stdlib.h as unsigned int. On success, `malloc()` returns a pointer to the first byte of allocated memory. The returned pointer is of type `void`, which can be type cast to appropriate type of pointer. It is generally used as-

```
ptr = (datatype *)malloc(specified size);
```

Here `ptr` is a pointer of type `datatype`, and specified size is the size in bytes required to be reserved in memory. The expression `(datatype *)` is used to typecast the pointer returned by `malloc()`. For example-

```
int *ptr;
ptr = (int *)malloc(12);
```

This allocates 12 contiguous bytes of memory space and the address of first byte is stored in the pointer variable `ptr`. The allocated memory contains garbage value. We can use `sizeof` operator to make the program portable and more readable.

```
ptr = (int *)malloc(5*sizeof(int));
```

This allocates the memory space to hold five integer values.

If there is not sufficient memory available in heap then `malloc()` returns NULL. So we should always check the value returned by `malloc()`.

```
ptr = (float *)malloc(10*sizeof(float));
if(ptr==NULL)
        printf("Sufficient memory not available");
```

Unlike memory allocated for variables and arrays, dynamically allocated memory has no name associated with it. So it can be accessed only through pointers. We have a pointer which points to the first byte of the allocated memory and we can access the subsequent bytes using pointer arithmetic.

```
/*P2.20 Program to understand dynamic allocation of memory*/
#include<stdio.h>
#include<stdlib.h>
main()
{
        int *p,n,i;
        printf("Enter the number of integers to be entered : ");
        scanf("%d",&n);
        p = (int *)malloc(n*sizeof(int));
        if(p==NULL)
        {
                printf("Memory not available\n");
                exit(1);
        }
        for(i=0; i<n; i++)
        {
```

```
            printf("Enter an integer : ");
            scanf("%d",p+i);
        }
        for(i=0; i<n; i++)
            printf("%d\t",*(p+i));
}
```

The function `malloc()` returns a void pointer and we have studied that a void pointer can be assigned to any type of pointer without typecasting. But we have used typecasting because it is a good practice to do so and moreover it also ensures compatibility with C++.

2.3.2 calloc( )

Declaration : `void *calloc(size_t n, size_t size);`

The `calloc()` function is used to allocate multiple blocks of memory. It is similar to `malloc()` function except for two differences. The first one is that it takes two arguments. The first argument specifies the number of blocks and the second one specifies the size of each block. For example-

```
ptr=(int *)calloc(5,sizeof(int));
```

This allocates 5 blocks of memory, each block contains 4 bytes and the starting address is stored in the pointer variable `ptr`, which is of type `int *`. An equivalent `malloc()` call would be-

```
ptr=(int *)malloc(5*sizeof(int));
```

Here we have to do the calculation ourselves by multiplying, but `calloc()` function does the calculation for us.

The other difference between `calloc()` and `malloc()` is that the memory allocated by `malloc()` contains garbage value while the memory allocated by `calloc()` is initialized to zero. But this initialization by `calloc()` is not very reliable, so it is better to explicitly initialize the elements whenever there is a need to do so.

Like `malloc()`, `calloc()` also returns NULL if there is not sufficient memory available in the heap

2.3.3 realloc( )

Declaration : `void *realloc(void *ptr, size_t newsize)`

We may want to increase or decrease the memory allocated by `malloc()` or `calloc()`. The function `realloc()` is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

This function takes two arguments, first is a pointer to the block of memory that was previously allocated by `malloc()` or `calloc()` and second one is the new size for that block. For example-

```
ptr = (int *)malloc(size);
```

This statement allocates the memory of the specified size and the starting address of this memory block is stored in the pointer variable `ptr`. If we want to change the size of this memory block, then we can use `realloc()` as-

```
ptr = (int *)realloc(ptr,newsize);
```

This statement allocates the memory space of `newsize` bytes, and the starting address of this memory block is stored in the pointer variable `ptr`. The `newsize` may be smaller or larger than the old size. If the `newsize` is larger, then the old data is not lost and the newly allocated bytes are uninitialized. The starting address contained in `ptr` may change if there is not sufficient memory at the old address to store all the bytes consecutively. This function moves the contents of old block into the new block and the data of the old block is not lost. On failure, `realloc()` returns NULL.

```
/*P2.21 Program to understand the use of realloc() function*/
#include<stdio.h>
```

```
#include<stdlib.h>
main()
{
        int i,*ptr;
        ptr = (int *)malloc(5*sizeof(int));
        if(ptr==NULL)
        {
                printf("Memory not available\n");
                exit(1);
        }
        printf("Enter 5 integers : ");
        for(i=0; i<5; i++)
                scanf("%d",ptr+i );
        /*Allocate memory for 4 more integers*/
        ptr = (int *)realloc(ptr,9*sizeof(int));
        if(ptr == NULL)
        {
                printf("Memory not available\n");
                exit(1);
        }
        printf("Enter 4 more integers : ");
        for(i=5; i<9; i++)
                scanf("%d",ptr+i);
        for(i=0; i<9; i++)
                printf("%d ",*(ptr+i));
}
```

2.3.4 free( )

Declaration : `void free(void *p)`

The dynamically allocated memory is not automatically released; it will exist till the end of program. If we have finished working with the memory allocated dynamically, it is our responsibility to release that memory so that it can be reused. The function `free()` is used to release the memory space allocated dynamically. The memory released by `free()` is made available to the heap again and can be used for some other purpose. For example-

```
free(ptr);
```

Here `ptr` is a pointer variable that contains the base address of a memory block created by `malloc()` or `calloc()`. Once a memory location is freed it should not be used. We should not try to free any memory location that was not allocated by `malloc()`, `calloc()` or `realloc()`.

When the program terminates, all the memory is released automatically by the operating system but it is a good practice to free whatever has been allocated dynamically. We won't get any errors if we don't free the dynamically allocated memory, but this would lead to memory leak i.e. memory is slowly leaking away and can be reused only after the termination of program. For example consider this function-

```
void func()
{
        int *ptr;
        ptr = (int*)malloc(10*sizeof(int));
        .....
}
```

Here we have allocated memory for 10 integers through `malloc()`, so each time this function is called, space for 10 integers would be reserved. We know that the local variables vanish when the function terminates, and since `ptr` is a local pointer variable, it will be deallocated automatically at the termination of function. But the space allocated dynamically is not deallocated automatically, so that space remains there and can't be used, leading to memory leaks. We should free the memory space by putting a call to `free()` at the end of the function.

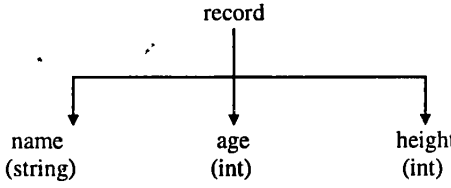
Since the memory space allocated dynamically is not released after the termination of function, it is valid to return a pointer to dynamically allocated memory. For example-

```
int *func()
{
        int *ptr;
        ptr = (int*)malloc(10*sizeof(int));
        .....
        return ptr;
}
```

Here we have allocated memory through `malloc()` in `func()`, and returned a pointer to this memory. Now the calling function receives the starting address of this memory, so it can use this memory. Note that now the call to function `free()` should be placed in the calling function when it has finished working with this memory. Here `func()` is declared as a function returning pointer. Recall that it is not valid to return address of a local variable since it vanishes after the termination of function.

2.4 Structure

Array is a collection of same type of elements but in many real life applications we may need to group different types of logically related data. For example if we want to create a record of a person that contains name, age and height of that person, then we can't use array because all the three data elements are of different types.



To store these related fields of different data types we can use a structure, which is capable of storing heterogeneous data. Data of different types can be grouped together under a single name using structures. The data elements of a structure are referred to as members.

2.4.1 Defining a Structure

Definition of a structure creates a template or format that describes the characteristics of its members. All the variables that would be declared of this structure type, will take the form of this template. The general syntax of a structure definition is-

```
struct tagname{
                datatype member1;
                datatype member2;
                .....
                .....
                datatype memberN;
               };
```

Here `struct` is a keyword, which tells the compiler that a structure is being defined. `member1`, `member2`, ..., `memberN` are members of the structure and are declared inside curly braces. There should be a semicolon at the end of the curly braces. These members can be of any data type like int, char, float, array, pointer or another structure type. `tagname` is the name of the structure and it is used further in the program to declare variables of this structure type.

Definition of a structure provides one more data type in addition to the built in data types. We can declare variables of this new data type that will have the format of the defined structure. It is important to note that definition of a structure template does not reserve any space in memory for the members; space is reserved only when actual variables of this structure type are declared. Although the syntax of declaration of members inside the template is identical to the syntax we use in declaring variables, these members are not variables, they don't

have any existence until they are attached with a structure variable. The member names inside a structure should be different but these names can be similar to any other variable name declared outside the structure. The member names of two different structures may also be same. Let us take an example of defining a structure template.

```
struct student{
                    char name[20];
                    int rollno;
                    float marks;
                };
```

Here `student` is the structure tag and there are three members of this structure viz `name`, `rollno` and `marks`. Structure template can be defined globally or locally i.e. it can be placed before all functions in the program or it can be locally present in a function. If the template is global then it can be used by all functions while if it is local then only the function containing it can use it.

2.4.2 Declaring Structure Variables

By defining a structure we have only created a template or format, the actual use of structures will be when we declare variables based on this template. We can declare structure variables in two ways-

1. With structure definition
2. Using the structure tag

2.4.2.1 With Structure Definition

```
struct student{
                    char name[20];
                    int rollno;
                    float marks;
                }stu1,stu2,stu3;
```

Here `stu1`, `stu2` and `stu3` are variables of type `struct student`. When we declare a variable while defining the structure template, the tagname is optional. So we can also declare them as-

```
struct{
          char name[20];
          int rollno;
          float marks;
      }stu1,stu2,stu3;
```

If we declare variables in this way, then we'll not be able to declare other variables of this structure type anywhere else in the program nor can we send these structure variables to functions. If a need arises to declare a variable of this type in the program then we'll have to write the whole template again. So although the tagname is optional it is always better to specify a tagname for the structure.

2.4.2.2 Using Structure Tag

We can also declare structure variables using structure tag. This can be written as-

```
struct student{
                    char name[20];
                    int rollno;
                  float marks;
                 };
struct student stu1,stu2;
struct student stu3;.
```

Here `stu1`, `stu2` and `stu3` are structure variables that are declared using the structure tag `student`. Declaring a structure variable reserves space in memory. Each structure variable declared to be of type `struct student` has three members viz. `name`, `rollno` and `marks`. The compiler will reserve space for each variable sufficient to hold all the members.

2.4.3 Initialization of Structure Variables

The syntax of initializing structure variables is similar to that of arrays. All the values are given in curly braces and the number, order and type of these values should be same as in the structure template definition. The initializing values can only be constant expressions.

```
struct student{
                char name[20];
                int rollno;
                float marks;
             }stu1 = {"Mary",25,98};
struct student stu2 = {"John",24,67.5};
```

Here value of members of stu1 will be "Mary" for name, 25 for rollno, 98 for marks. The values of members of stu2 will be "John" for name, 24 for rollno, 67.5 for marks.

We cannot initialize members while defining the structure.

```
struct student {
                    char name[20];
                    int rollno;
                    float marks = 99;  /*Invalid*/
                 }stu;
```

This is invalid because there is no variable called marks, and no memory is allocated for structure definition.
If the number of initializers is less than the number of members then the remaining members are initialized with zero. For example if we have this initialization-

```
struct student stu1 = {"Mary"};
```

Here the members rollno and marks of stu1 will be initialized to zero. This is equivalent to the initialization-

```
struct student stu1 = {"Mary",0,0};
```

2.4.4 Accessing Members of a Structure

For accessing any member of a structure variable, we use the dot (.) operator which is also known as the period or membership operator. The format for accessing a structure member is-

```
structvariable.member
```

Here on the left side of the dot there should be a variable of structure type and on right hand side there should be the name of a member of that structure. For example consider the following structure-

```
struct student {
                    char name[20];
                    int rollno;
                    float marks;
                 };
struct student stu1,stu2;
```

name of stu1 is given by - stu1.name
rollno of stu1 is given by - stu1.rollno
marks of stu1 is given by - stu1.marks
name of stu2 is given by - stu2.name
rollno of stu2 is given by - stu2.rollno
marks of stu2 is given by - stu2.marks

We can use stu1.name, stu1.marks, stu2.marks etc like any other ordinary variables in the program. They can be read, displayed, processed, assigned values or can be sent to functions as arguments. We can't use student.name or student.rollno because student is not a structure variable, it is a structure tag.

```
/*P2.22 Program to display the values of structure members*/
#include<stdio.h>
#include<string.h>
```

```
struct student {
                char name[20];
                int rollno;
                float marks;
              };
main()
{
        struct student stu1 = {"Mary",25,68};
        struct student stu2,stu3;
        strcpy(stu2.name,"John");
        stu2.rollno = 26;
        stu2.marks = 98;
        printf("Enter name, rollno and marks for stu3 : ");
        scanf("%s %d %f",stu3.name,&stu3.rollno,&stu3.marks);
        printf("stu1 : %s  %d  %.2f\n",stu1.name,stu1.rollno,stu1.marks);
        printf("stu2 : %s  %d  %.2f\n",stu2.name,stu2.rollno,stu2.marks);
        printf("stu3 : %s  %d  %.2f\n",stu3.name,stu3.rollno,stu3.marks);
}
```

In this program we have declared three variables of type `struct student`. The first variable `stu1` has been initialized, the members of second variable `stu2` are given values using separate statements and the values for third variable `stu3` are input by the user.

The dot operator is one of the highest precedence operators; its associativity is from left to right. Hence it will take precedence over all other unary, relational, logical, arithmetic and assignment operators. So in an expression like `++stu.marks`, first `stu.marks` will be accessed and then its value will be increased by 1.

2.4.5 Assignment of Structure Variables

We can assign values of a structure variable to another structure variable, if both variables are of the same structure type. For example-

```
/*P2.23 Program to assign a structure variable to another structure variable*/
#include<stdio.h>
struct student {
                char name[20];
                int rollno;
                float marks;
              };
main()
{
        struct student stu1 = {"Oliver",12,98},stu2;
        stu2 = stu1;
        printf("stu1 : %s   %d   %.2f\n",stu1.name,stu1.rollno,stu1.marks);
        printf("stu2 : %s   %d   %.2f\n",stu2.name,stu2.rollno,stu2.marks);
}
```

Unary, relational, arithmetic, bitwise operators are not allowed with structure variables. We can use these variables with the members provided the member is not a structure itself.

2.4.6 Array of Structures

We know that array is a collection of elements of same datatype. We can declare array of structures where each element of array is of structure type. Array of structures can be declared as-

```
struct student stu[10];
```

Here `stu` is an array of 10 elements, each of which is a structure of type `struct student`, means each element of `stu` has 3 members, which are `name`, `rollno` and `marks`. These structures can be accessed through subscript notation. To access the individual members of these structures we'll use the dot operator as usual.

```
stu[0].name     stu[0].rollno     stu[0].marks
stu[1].name     stu[1].rollno     stu[1].marks
stu[2].name     stu[2].rollno     stu[2].marks
.....           .....             .....
.....           .....             .....
stu[9].name     stu[9].rollno     stu[9].marks
```

All the structures of an array are stored in consecutive memory locations.

```
/*P2.24 Program to understand array of structures*/
#include<stdio.h>
struct student {
                char name[20];
                int rollno;
                float marks;
              };
main()
{
        int i;
        struct student stuarr[10];
        for(i=0; i<10; i++)
        {
           printf("Enter name,rollno and marks : ");
           scanf("%s%d%f",stuarr[i].name,&stuarr[i].rollno,&stuarr[i].marks);
        }
        for(i=0; i<10; i++)
           printf("%s %d %f\n",stuarr[i].name,stuarr[i].rollno,stuarr[i].marks);
}
```

An array of structure can be initialized as-

```
struct student stuarr[3] = {
                                {"Mary",12,98.5},
                                {"John",11,97},
                                {"Tom",12,89.5}
                            };
```

The inner pairs of braces are optional if all the initializers are present in the list.

2.4.7 Arrays within Structures

We can have an array as a member of structure. In structure `student`, we have taken the member `name` as an array of characters. Now we'll declare another array inside the structure `student`.

```
struct student {
                  char name[20];
                  int rollno;
                  int submarks[4];
                };
```

The array `submarks` denotes the marks of students in 4 subjects.

If `stu` is a variable of type `struct student` then-

`stu.submarks[0]` - Denotes the marks of the student in first subject
`stu.submarks[1]` - Denotes the marks in second subject.
`stu.name[0]` - Denotes the first character of the name member.
`stu.name[4]` - Denotes the fifth character of the name member.

If `stuarr` is an array of size 10 of type `struct student` then-

`stuarr[0].submarks[0]` - Denotes the marks of first student in first subject
`stuarr[4].submarks[3]` - Denotes the marks of fifth student in fourth subject.
`stuarr[0].name[0]` - Denotes the first character of name member of first student
`stuarr[5].name[7]` - Denotes the eighth character of name member of sixth student

```
/*P2.25 Program to understand arrays within structures*/
#include<stdio.h>
struct student {
                    char name[20];
                    int rollno;
                    int submarks[4];
                 };
main()
{
    int i,j;
    struct student stuarr[3];

    for(i=0; i<3; i++)
    {
        printf("Enter data for student %d\n",i+1);
        printf("Enter name : ");
        scanf("%s",stuarr[i].name);
        printf("Enter roll number : ");
        scanf("%d",&stuarr[i].rollno);
        for(j=0; j<4; j++)
        {
                printf("Enter marks for subject %d : ",j+1);
                scanf("%d",&stuarr[i].submarks[j]);
        }
    }
    for(i=0; i<3; i++)
    {
        printf("Data of student %d\n",i+1);
        printf("Name:%s,Roll number:%d\nMarks:",stuarr[i].name,stuarr[i].rollno);
        for(j=0; j<4; j++)
                printf("%d   ",stuarr[i].submarks[j]);
        printf("\n");
    }
}
```

2.4.8 Nested Structures (Structure within Structure)

The members of a structure can be of any data type including another structure type i.e. we can include a structure within another structure. A structure variable can be a member of another structure. This is called nesting of structures.

```
struct tag1{
                member1;
                member2;
                ..........
                struct tag2{
                                member1;
                                member2;
                                ..........
                                member m;
                              }var1;
                ..........
                member n;
           }var2;
```

For accessing member1 of inner structure we will write -

```
var2.var1.member1
```

Here is an example of a nested structure -

```
struct student{
                char name[20];
                int rollno;
                struct date{
                        int day;
                        int month;
                        int year;
                        }birthdate;
                float marks;
        }stu1,stu2;
```

Here we have defined a structure `date` inside the structure `student`. This structure `date` has three members `day`, `month`, `year` and `birthdate` is a variable of type `struct date`. We can access the members of inner structure as-

`stu1.birthdate.day` → day of birthdate of stu1
`stu1.birthdate.month` → month of birthdate of stu1
`stu1.birthdate.year` → year of birthdate of stu1
`stu2.birthdate.day` → day of birthdate of stu2

Here we have defined the template of structure date inside the structure `student`, we could have defined it outside and declared its variables inside the structure student using the tag. But remember if we define the inner structure outside, then this definition should always be before the definition of outer structure. Here in this case the date structure should be defined before the student structure.

```
struct date{
                int day;
                int month;
                int year;
             };
struct  student{
                    char name[20];
                    int rollno;
                    float marks;
                    struct date birthdate;
                 }stu1,stu2;
```

The advantage of defining `date` structure outside is that we can declare variables of `date` type anywhere else also. Suppose we define a structure `teacher`, then we can declare variables of `date` structure inside it as-

```
struct teacher{
                    char name[20];
                    int age;
                    float salary;
                    struct date birthdate, joindate;
                }t1,t2;
```

The nested structures may also be initialized at the time of declaration. For example-

```
struct teacher t1 = {"Sam",34,9000,{8,12,1970},{1,7,1995}};
```

Nesting of a structure within itself is not valid. For example the following structure definition is invalid-

```
struct person{
                  char name[20];
                  int age;
                  float height;
                  struct person father;  /*Invalid*/
              }emp;
```

The nesting of structures can be extended to any level. The following example shows nesting at level three i.e. first structure is nested inside a second structure and second structure is nested inside a third structure.

```
struct time
{
        int hr;
        int min;
```

```
        int sec;
};
struct date
{
        int day;
        int month;
        int year;
        struct time t;
};
struct student
{
        char name[20];
        struct date dob;        /*Date of birth*/
}stu1,stu2;
```

To access hour of date of birth of student `stu1` we can write-

```
stu1.dob.t.hr
```

2.4.9 Pointers to Structures

We have studied that pointer is a variable which holds the starting address of another variable of any data type like int, float or char. Similarly we can have pointer to structure, which can point to the starting address of a structure variable. These pointers are called structure pointers and can be declared as-

```
struct student{
                    char name[20];
                    int rollno;
                    int marks;
                };
struct student stu,*ptr;
```

Here `ptr` is a pointer variable that can point to a variable of type `struct student`. We will use the & operator to access the starting address of a structure variable, so `ptr` can point to `stu` by-

```
ptr = &stu;
```

There are two ways of accessing the members of structure through the structure pointer.

We know `ptr` is a pointer to a structure, so by dereferencing it we can get the contents of structure variable. Hence `*ptr` will give the contents of `stu`. So to access members of a structure variable `stu` we can write-

```
(*ptr).name
(*ptr).rollno
(*ptr).marks
```

Here parentheses are necessary because dot operator has higher precedence than the * operator. This syntax is confusing so C has provided another facility of accessing structure members through pointers. We can use the arrow operator (->) which is formed by hyphen symbol and greater than symbol. We can access the members as-

```
ptr->name
ptr->rollno
ptr->marks
```

The arrow operator has same precedence as that of dot operator and it also associates from left to right.

```
/*P2.26 Program to understand pointers to structures*/
#include<stdio.h>
struct student{
                    char name[20];
                    int rollno;
                    int marks;
                };
main()
{
        struct student stu = {"Mary",25,68};
```

```
        struct student *ptr = &stu;
        printf("Name - %s\t",ptr->name);
        printf("Rollno - %d\t",ptr->rollno);
        printf("Marks - %d\n",ptr->marks);
}
```

We can also have pointers that point to individual members of a structure variable. For example-

```
int *p = &stu.rollno;
float *ptr = &stu.marks;
```

The expression `&stu.rollno` is equivalent to `&(stu.rollno)` because the precedence of dot operator is more than that of address operator.

2.4.10 Pointers within Structures

A pointer can also be used as a member of structure. For example we can define a structure like this-

```
struct student{
                    char name[20];
                    int *ptrmem;
                };
struct student stu,*stuptr = &stu;
```

Here `ptrmem` is pointer to int and is a member of the structure `student`.

To access the value of `ptrmem`, we'll write-

`stu.ptrmem` or `stuptr->ptrmem`

To access the value pointed to by `stu.ptrmem`, we'll write-

`*stu.ptrmem` or `*stuptr->ptrmem`

Since the priority of dot and arrow operators is more than that of dereference operator, the expression `*stu.ptrmem` is equivalent to `*(stu.ptrmem)`, and the expression `*stuptr->ptrmem` is equivalent to `*(stuptr->ptrmem)`.

2.4.11 Structures and Functions

Structures may be passed as arguments to function in different ways. We can pass individual members, whole structure variable or structure pointers to the function. Similarly a function can return either a structure member or whole structure variable or a pointer to structure.

2.4.11.1 Passing Structure Members as Arguments

We can pass individual structure members as arguments to functions like any other ordinary variable.

```
/*P2.27 Program to understand how structure members are sent to a function*/
#include<stdio.h>
#include<string.h>
struct student{
                    char name[20];
                    int rollno;
                    int marks;
                };
void display(char name[],int rollno,int marks);
main()
{
        struct student stu1 = {"John",12,87};
        struct student stu2;
        strcpy(stu2.name,"Mary");
        stu2.rollno = 18;
        stu2.marks = 90;
```

```
        display(stu1.name,stu1.rollno,stu1.marks);
        display(stu2.name,stu2.rollno,stu2.marks);
}
void display(char name[],int rollno,int marks)
{
        printf("Name   - %s\t",name);
        printf("Rollno - %d\t",rollno);
        printf("Marks  - %d\n",marks);
}
```

Here we have passed members of the variables `stu1` and `stu2` to the function `display()`. We can pass the arguments using call by reference also so that the changes made in the called function will be reflected in the calling function. In that case we'll have to send the addresses of the members. It is also possible to return a single member from a function.

2.4.11.2 Passing Structure Variable as Argument

Passing individual members to function becomes cumbersome when there are many members and the relationship between the members is also lost. We can pass the whole structure as an argument.

```
/*P2.28 Program to understand how a structure variable is sent to a function*/
#include<stdio.h>
struct student {
                    char name[20];
                    int rollno;
                    int marks;
                };
void display(struct student);
main()
{
        struct student stu1 = {"John",12,87};
        struct student stu2 = {"Mary",18,90};
        display(stu1);
        display(stu2);
}
void display(struct student stu)
{
        printf("Name   - %s\t", stu.name);
        printf("Rollno - %d\t", stu.rollno);
        printf("Marks  - %d\n", stu.marks);
}
```

Here it is necessary to define the structure template globally because it is used by both functions to declare variables.

The name of a structure variable is not a pointer unlike arrays, so when we send a structure variable as an argument to a function, a copy of the whole structure is made inside the called function and all the work is done on that copy. Any changes made inside the called function are not visible in the calling function since we are only working on a copy of the structure variable, not on the actual structure variable.

2.4.11.3 Passing Pointers to Structures as Arguments

If the size of a structure is very large, then it is not efficient to pass the whole structure to the function since a copy of it has to be made inside the called function. In this case it is better to send address of the structure, which will improve the execution speed.

We can access the members of the structure variable inside the calling function using arrow operator. In this case any changes made to the structure variable inside the called function, will be visible in the calling function since we are actually working on the original structure variable.

```
/*P2.29 Program to understand how a pointer to structure variable is sent to a function*/
#include<stdio.h>
struct student{
```

```
                char name[20];
                int rollno;
                int marks;
            };
void display(struct student *);
void inc_marks(struct student *);
main()
{
        struct student stu1 = {"John",12,87};
        struct student stu2 = {"Mary",18,90};
        inc_marks(&stu1);
        inc_marks(&stu2);
        display(&stu1);
        display(&stu2);
}
void inc_marks(struct student *stuptr)
{
        (stuptr->marks)++;
}
void display(struct student *stuptr)
{
        printf("Name   - %s\t",stuptr->name);
        printf("Rollno - %d\t",stuptr->rollno);
        printf("Marks  - %d\n",stuptr->marks);
}
```

2.4.11.4 Returning a Structure Variable from Function

Structure variables can be returned from functions as any other variable. The returned value can be assigned to a structure of the appropriate type.

```
/*P2.30 Program to understand how a structure variable is returned from a function*/
#include<stdio.h>
struct student{
                char name[20];
                int rollno;
                int marks;
            };
void display(struct student);
struct student change(struct student stu);
main()
{
        struct student stu1 = {"John",12,87};
        struct student stu2 = {"Mary",18,90};
        stu1 = change(stu1);
        stu2 = change(stu2);
        display(stu1);
        display(stu2);
}
struct student change(struct student stu)
{
        stu.marks = stu.marks + 5;
        stu.rollno = stu.rollno - 10;
        return stu;
}
void display(struct student stu)
{
        printf("Name   - %s\t",stu.name);
        printf("Rollno - %d\t",stu.rollno);
        printf("Marks  - %d\n",stu.marks);
}
```

2.4.11.5 Returning a Pointer to Structure from a Function

Pointers to structures can also be returned from functions. In the following program, the function `func()` returns a pointer to structure.

```
/*P2.31 Program to understand how a pointer to structure is returned from a function*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct student{
                    char name[20];
                    int rollno;
                    int marks;
                };
void display(struct student *);
struct student *func();
struct student *ptr;
main()
{
        struct student *stuptr;
        stuptr = func();
        display(stuptr);
        free(stuptr);
}
struct student *func()
{
        ptr = (struct student *)malloc(sizeof(struct student));
        strcpy(ptr->name,"Joseph");
        ptr->rollno = 15;
        ptr->marks = 98;
        return ptr;
}
void display(struct student *stuptr)
{
        printf("Name   - %s\t",stuptr->name);
        printf("Rollno  - %d\t",stuptr->rollno);
        printf("Marks   - %d\n",stuptr->marks);
}
```

2.4.11.6 Passing Array of Structures as Argument

We can pass an array of structure to function, where each element of array is of structure type. This can be written as-

```
/*P2.32 Program to understand how an array of structures is sent to a function*/
#include<stdio.h>
struct student{
                    char name[20];
                    int rollno;
                    int marks;
                };
void display(struct student);
void dec_marks(struct student stuarr[]);
main()
{
        int i;
        struct student stuarr[3] = {
                                        {"Mary", 12, 98},
                                        {"John", 11, 97},
                                        {"Tom", 12, 89}
                                    };
        dec_marks(stuarr);
        for(i=0; i<3; i++)
                display(stuarr[i]);
```

```
}
void dec_marks(struct student stuarr[])
{
        int i;
        for(i=0; i<3; i++)
                stuarr[i].marks = stuarr[i].marks-10;
}
void display(struct student stu)
{
        printf("Name  - %s\t", stu.name);
        printf("Rollno  - %d\t", stu.rollno);
        printf("Marks  - %d\n", stu.marks);
}
```

All the changes made in the array of structures inside the called function will be visible in the calling function.

2.4.11.7 Self Referential Structures

A structure that contains pointers to structure of its own type is known as self referential structure. For example-

```
struct tag{
              datatype member1;
              datatype member2;
              .....
              .....
              struct tag *ptr1;
              struct tag *ptr2;
          };
```

Here `ptr1` and `ptr2` are structure pointers that can point to structure variables of type `struct tag`, so `struct tag` is a self referential structure. These types of structures are helpful in implementing data structures like linked lists and trees.

Exercise

Find the output of the following programs.

(1)

```
main()
{
    int i,size=5,arr[size];
    for(i=0; i<size; i++)
            scanf("%d",&arr[i]);
    for(i=0; i<size; i++)
            printf("%d  ",arr[i]);
}
```

(2)

```
main()
{
    int arr[4]={2,4,8,16},i=4,j;
    while(i)
    {
            j = arr[i] + i;
            i--;
    }
    printf("j = %d\n",j);
}
```

(3)

```
main()
{
    int i=0,sum=0,arr[6]={4,2,6,0,5,10};
    while(arr[i])
    {
```

```
            sum = sum+arr[i];
            i++;
        }
        printf("sum = %d\n",sum);
    }
```

(4)

```
    void func(int arr[]);
    main()
    {
        int arr[5] = {5,10,15,20,25};
        func(arr);
    }
    void func(int arr[])
    {
        int i=5,sum=0;
        while(i>2)
                sum = sum+arr[--i];
        printf("sum = %d\n",sum);
    }
```

(5)

```
    void swapvar(int a,int b);
    void swaparr(int arr1[5],int arr2[5]);
    main()
    {
        int a=4,b=6;
        int arr1[5] = {1,2,3,4,5};
        int arr2[5] = {6,7,8,9,10};
        swapvar(a,b);
        swaparr(arr1,arr2);
        printf("a = %d, b = %d\n",a,b);
        printf("arr1[0] = %d, arr1[4] = %d\n",arr1[0],arr1[4]);
        printf("arr2[0] = %d, arr2[4] = %d\n",arr2[0],arr2[4]);
    }
    void swapvar(int a,int b)
    {
        int temp;
        temp=a, a=b, b=temp;
    }
    void swaparr(int arr1[5],int arr2[5])
    {
        int i,temp;
        for(i=0; i<5; i++)
        { temp=arr1[i], arr1[i]=arr2[i], arr2[i]=temp; }
    }
```

(6)

```
    main()
    {
        int i,arr[5]={25,30,35,40,45},*p;
        p = arr;
        for(i=0; i<5; i++)
                printf("%d\t%d\t",*(p+i),p[i]);
    }
```

(7)

```
    main()
    {
        int i,arr[5]={25,30,35,40,55};
        for(i=0; i<5; i++)
        {
                printf("%d    ", *arr);
                arr++;
        }
    }
```

(8)
```
main()
{
   int i,arr[5] = {25,30,35,40,45},*p = arr;
   for(i=0; i<5; i++)
   {
      (*p)++;
      printf("%d ",*p);
      p++;
   }
}
```

(9)
```
main()
{
   int arr[5]={25,30,35,40,55},*p;
   for(p=&arr[0]; p<arr+5; p++)
          printf("%d   ",*p);
}
```

(10)
```
main()
{
   int arr[10]={25,30,35,40,55,60,65,70,85,90},*p;
   for(p=arr+2; p<arr+8; p=p+2)
          printf("%d  ",*p);
}
```

(11)
```
main()
 {
   int i, arr[10]={25,30,35,40,55,60,65,70,85,90};
   int *p = arr+9;
   for(i=0; i<10; i++)
          printf("%d   ",*p--);
 }
```

(12)
```
main()
{
   int arr[10] = {25,30,35,40,55,60,65,70,85,90},*p;
   for(p=arr+9; p>=arr; p--)
          printf("%d   ",*p);
}
```

(13)
```
 int a=5, b=10;
change1(int *p);
change2(int **pp);
main()
{
     int x=20, *ptr=&x;
     printf("%d  ",*ptr);
     change1(ptr);
     printf("%d  ",*ptr);
     change2(&ptr);
     printf("%d\n",*ptr);
}
change1(int *p)
{
     p = &a;
}
change2(int **pp)
{
     *pp = &b;
}
```

(14)
```
void func(int x,int *y);
```

```
main()
{
     int a=2,b=6;
     func(a,&b);
     printf("a = %d, b = %d\n",a,b);
}
void func(int x,int *y)
{
     int temp;
     temp = x;
     x = *y;
     *y = temp;
}
```

(15)

```
void func(int a[]);
main()
{
     int arr[10] = {1,2,3,4,5,6,7,8,9,10};
     func(arr+3);
}
void func(int a[])
{
     int i;
     for(i=0; a[i]!=8; i++)
          printf("%d ",a[i]);
}
```

(16)

```
main()
{
     struct A {
                    int marks;
                    char grade;
               };
     struct A A1,B1;
     A1.marks = 80;
     A1.grade = 'A';
     printf("Marks = %d\t",A1.marks);
     printf("Grade = %c\t",A1.grade);
     B1 = A1;
     printf("Marks = %d\t",B1.marks);
     printf("Grade = %c\n",B1.grade);
}
```

(17)

```
void func(struct tag v);
main()
{
     struct tag{
                 int i;
                 char c;
               };
     struct tag var = {2,'s'};
     func(var);
}
void func(struct tag v)
{
     printf("%d  %c\n",v.i,v.c);
}
```

(18)

```
struct tag{int i; char c;};
void func(struct tag);
main()
{
     struct tag var = {12,'c'};
```

```
    func(var);
    printf("%d\n",var.i);
}
void func(struct tag var)
{
    var.i++;
}
```

(19)

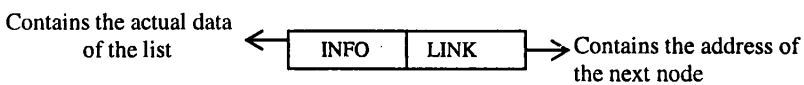
```
struct tag{int i; char c;};
void func(struct tag *);
main()
{
    struct tag var = {12,'c'};
    func(&var);
    printf("%d\n",var.i);
}
void func(struct tag *ptr)
{
    ptr->i++;
}
```

Linked Lists

List is a collection of similar type of elements. There are two ways of maintaining a list in memory. The first way is to store the elements of the list in an array, but arrays have some restrictions and disadvantages. The second way of maintaining a list in memory is through linked list. Now let us study what a linked list is and after that we will come to know how it overcomes the limitations of array.

3.1 Single Linked list

A single linked list is made up of nodes where each node has two parts, the first one is the info part that contains the actual data of the list and the second one is the link part that points to the next node of the list or we can say that it contains the address of the next node.



The beginning of the list is marked by a special pointer named `start`. This pointer points to the first node of the list. The link part of each node points to the next node in the list, but the link part of last node has no next node to point to, so it is made NULL. Hence if we reach a node whose link part has NULL value then we know that we are at the end of the list. Suppose we have a list of four integers 33, 44, 55, 66; let us see how we will represent it through linked list.

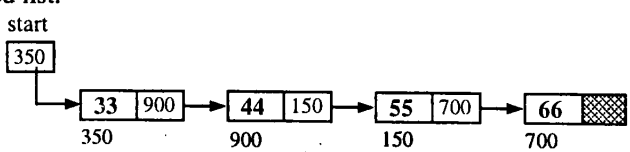


Figure 3.1 Single Linked List

From the figure it is clear that the info part contains the integer values and the link part contains the address of the next node. The address of the first node is contained in the pointer `start` and the link part of last node is NULL(represented by shaded part in the figure).

If we observe the memory addresses of the nodes, we find that the nodes are not necessarily located adjacent to each other in the memory. The following figure shows the position of nodes of the above list in memory.

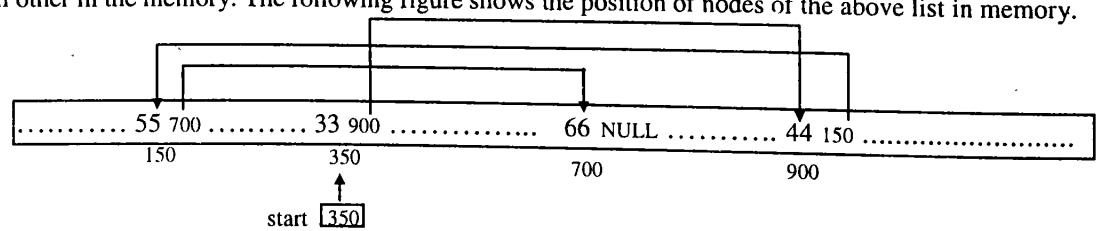


Figure 3.2

We can see that the nodes are scattered here and there in memory, but still they are connected to each other through the link part, which also maintains their linear order. Now let us see the picture of memory if the same list of integers is implemented through array.

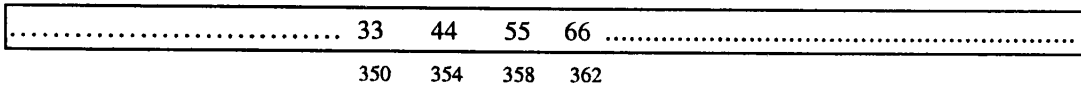


Figure 3.3

Here the elements are stored in consecutive memory locations in the same order as they appear in the list. So array is sequential representation of list while linked list is the linked representation of list. In a linked list, nodes are not stored contiguously as in array, but they are linked through pointers(links) and we can use these links to move through the list.

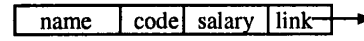
Now let us see how we can represent a node of a single linked list in C language. We will take a self referential structure for this purpose. Recall that a self referential structure is a structure which contains a pointer to a structure of the same type. The general form of a node of linked list is-

```
struct node{
            type1 member1;
            type2 member2;
            .....
            .....
            struct node *link;  /*Pointer to next node*/
        };
```

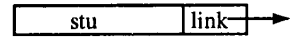
Let us see some examples of nodes-

```
struct node{
            int value;
            struct node *link;
        };
```

```
struct node{
            char name[10];
            int code;
            float salary;
            struct node *link;
        };
```




```
struct node{
            struct student stu;
            struct node *link;
        };
```



In this chapter we will perform all operations on linked lists that contain only an integer value in the info part of their nodes.

In array we could perform all the operations using the array name and index. In the case of linked list, we will perform all the operations with the help of the pointer `start` because it is the only source through which we can access our linked list. The list will be considered empty if the pointer `start` contains `NULL` value. So our first job is to declare the pointer `start` and initialize it to `NULL`. This can be done as-

```
struct node *start;
start = NULL;
```

Now we will discuss the following operations on a single linked list.

(i) Traversal of a linked list
(ii) Searching an element
(iii) Insertion of an element
(iv) Deletion of an element
(v) Creation of a linked list
(vi) Reversal of a linked list

The `main()` function and declarations are given here, and the code of other functions are given with the explanation.

```
/*P3.1 Program of single linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
struct node *create_list(struct node *start);
void display(struct node *start);
void count(struct node *start);
void search(struct node *start,int data);
struct node *addatbeg(struct node *start,int data);
struct node *addatend(struct node *start,int data);
struct node *addafter(struct node *start,int data,int item);
struct node *addbefore(struct node *start,int data,int item);
struct node *addatpos(struct node *start,int data,int pos);
struct node *del(struct node *start,int data);
struct node *reverse(struct node *start);

main()
{
        struct node *start=NULL;
        int choice,data,item,pos;
        while(1)
        {
                printf("1.Create List\n");
                printf("2.Display\n");
                printf("3.Count\n");
                printf("4.Search\n");
                printf("5.Add to empty list / Add at beginning\n");
                printf("6.Add at end\n");
                printf("7.Add after node\n");
                printf("8.Add before node\n");
                printf("9.Add at position\n");
                printf("10.Delete\n");
                printf("11.Reverse\n");
                printf("12.Quit\n\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        start = create_list(start);
                        break;
                 case 2:
                        display(start);
                        break;
                 case 3:
                        count(start);
                        break;
                 case 4:
                        printf("Enter the element to be searched : ");
                        scanf("%d",&data);
                        search(start,data);
                        break;
                 case 5:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start = addatbeg(start,data);
                        break;
```

```
            case 6:
                  printf("Enter the element to be inserted : ");
                  scanf("%d",&data);
                  start = addatend(start,data);
                  break;
            case 7:
                  printf("Enter the element to be inserted : ");
                  scanf("%d",&data);
                  printf("Enter the element after which to insert : ");
                  scanf("%d",&item);
                  start = addafter(start,data,item);
                  break;
            case 8:
                  printf("Enter the element to be inserted : ");
                  scanf("%d",&data);
                  printf("Enter the element before which to insert: ");
                  scanf("%d",&item);
                  start = addbefore(start,data,item);
                  break;
            case 9:
                  printf("Enter the element to be inserted : ");
                  scanf("%d",&data);
                  printf("Enter the position at which to insert : ");
                  scanf("%d",&pos);
                  start = addatpos(start,data,pos);
                  break;
            case 10:
                  printf("Enter the element to be deleted : ");
                  scanf("%d",&data);
                  start = del(start, data);
                  break;
            case 11:
                  start = reverse(start);
                  break;
            case 12:
                  exit(1);
            default:
                  printf("Wrong choice\n");
          }/*End of switch*/
      }/*End of while*/
}/*End of main()*/
```

In the function `main()`, we have taken an infinite loop and inside the loop we have written a switch statement. In the different cases of this switch statement, we have implemented different operations of linked list. To come out of the infinite loop and exit the program we have used the function `exit()`.

The structure pointer `start` is declared in `main()` and initialized to `NULL`. We send this pointer to all functions because `start` is the only way of accessing linked list. Functions like those of insertion, deletion, reversal will change our linked list and value of `start` might change so these functions return the value of `start`. The functions like `display()`, `count()`, `search()` do not change the linked list so their return type is `void`.

3.1.1 Traversing a Single Linked List

Traversal means visiting each node, starting from the first node till we reach the last node. For this we will take a structure pointer `p` which will point to the node that is currently being visited. Initially we have to visit the first node so `p` is assigned the value of `start`.

```
p = start;
```

Now `p` points to the first node of linked list. We can access the info part of first node by writing `p->info`. Now we have to shift the pointer `p` forward so that it points to the next node. This can be done by assigning the address of the next node to `p` as-

```
p = p->link;
```

Now p has address of the next node. Similarly we can visit each node of linked list through this assignment until p has NULL value, which is link part value of last element. So the linked list can be traversed as-

```
while(p!=NULL)
{
        printf("%d  ",p->info);
        p = p->link;
}
```

Let us take an example to understand how the assignment `p = p->link` makes the pointer p move forward. From now onwards we will not show the addresses, we will show only the info part of the list in the figures.

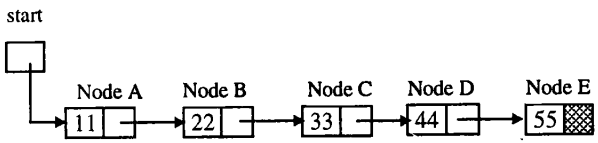


Figure 3.4

In figure 3.4, node A is the first node so start points to it, node E is the last node so its link is NULL.
Initially p points to node A, `p->info` gives 11 and `p->link` points to node B
After the statement `p = p->link;`
p points to node B, `p->info` gives 22 and `p->link` points to node C
After the statement `p = p->link;`
p points to node C, `p->info` gives 33 and `p->link` points to node D
After the statement `p = p->link;`
p points to node D, `p->info` gives 44 and `p->link` points to node E
After the statement `p = p->link;`
p points to node E, `p->info` gives 55 and `p->link` is NULL
After the statement `p = p->link;`
p becomes NULL, i.e. we have reached the end of the list so we come out of the loop.

The following function `display()` displays the contents of the linked list.

```
void display(struct node *start)
{
        struct node *p;
        if(start == NULL)
        {
                printf("List is empty\n");
                return;
        }
        p = start;
        printf("List is :\n");
        while(p != NULL)
        {
                printf("%d ",p->info);
                p = p->link;
        }
        printf("\n\n");
}/*End of display() */
```

Don't think of using start for moving forward. If we use `start = start->link`, instead of `p = p->link` then we will lose `start` and that is the only means of accessing our list. The following function `count()` finds out the number of elements of the linked list.

```
void count(struct node *start)
{
        struct node *p;
```

```
        int cnt = 0;
        p = start;
        while(p!=NULL)
        {
                p = p->link;
                cnt++;
        }
        printf("Number of elements are %d\n",cnt);
}/*End of count() */
```

3.1.2 Searching in a Single Linked List

For searching an element, we traverse the linked list and while traversing we compare the info part of each element with the given element to be searched. In the function given below, `item` is the element which we want to search.

```
void search(struct node *start,int item)
{
        struct node *p = start;
        int pos = 1;
        while(p!=NULL)
        {
                if(p->info == item)
                {
                        printf("Item %d found at position %d\n",item,pos);
                        return;
                }
                p = p->link;
                pos++;
        }
        printf("Item %d not found in list\n",item);
}/*End of search()*/
```

3.1.3 Insertion in a Single Linked List

There can be four cases while inserting a node in a linked list.

1. Insertion at the beginning.
2. Insertion in an empty list.
3. Insertion at the end.
4. Insertion in between the list nodes.

To insert a node, initially we will dynamically allocate space for that node using `malloc()`. Suppose `tmp` is a pointer that points to this dynamically allocated node. In the info part of the node we will put the data value.

```
tmp = (struct node *)malloc(sizeof(struct node));
tmp->info = data;
```

The link part of the node contains garbage value; we will assign address to it separately in the different cases. In our explanation we will refer to this new node as node T.

3.1.3.1 Insertion at the beginning of the list

We have to insert node T at the beginning of the list. Suppose the first node of list is node P, so the new node T should be inserted before it.

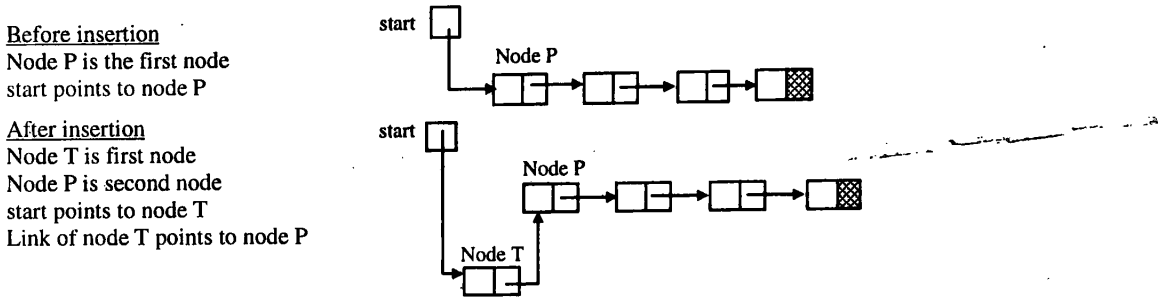


Figure 3.5 Insertion at the beginning of the list

(i) Link of node T should contain the address of node P, and we know that `start` has address of node P so we should write-

```
tmp->link = start;
```

After this statement, link of node T will point to node P.

(ii) We want to make node T the first node; hence we should update `start` so that now it points to node T.

```
start = tmp;
```

The order of the above two statements is important. First we should make link of T equal to `start` and after that only we should update `start`. Let's see what happens if the order of these two statements is reversed.

```
start = tmp;
```

`start` points to `tmp` and we lost the address of node P.

```
tmp->link = start;
```

Link of `tmp` will point to itself because `start` has address of `tmp`. So if we reverse the order, then link of node T will point to itself and we will be stuck in an infinite loop when the list is processed.

The following function `addatbeg()` adds a node at the beginning of the list.

```
struct node *addatbeg(struct node *start,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        tmp->link = start;
        start = tmp;
        return start;
}/*End of addatbeg()*/
```

3.1.3.2 Insertion in an empty list

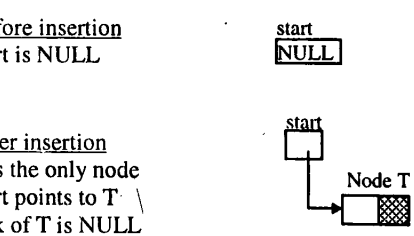


Figure 3.6 Insertion in an empty list

When the list is empty, value of `start` will be `NULL`. The new node that we are adding will be the only node in the list. Since it is the first node, `start` should point to this node and it is also the last node so its link should be `NULL`.

```
tmp->link = NULL;
start = tmp;
```

Since initially `start` was `NULL`, we can write `start` instead of `NULL` in the first statement, so now these two statements can be written as-

```
tmp->link = start;
start = tmp;
```

These two statements are same as in the previous case (3.1.3.1), so we can see that this case reduces to the previous case of insertion in the beginning and the same code can be written for both the cases.

3.1.3.3 Insertion at the end of the list

We have to insert a new node T at the end of the list. Suppose the last node of list is node P, so node T should be inserted after node P.

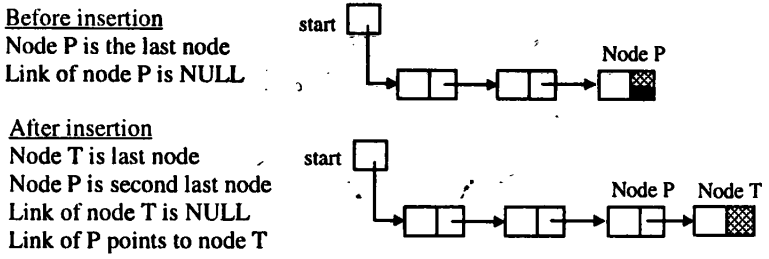


Figure 3.7 Insertion at the end of the list

Suppose we have a pointer `p` pointing to the node P. These are the two statements that should be written for this insertion-

```
p->link = tmp;
tmp->link = NULL;
```

So in this case we should have a pointer `p` pointing to the last node of the list. The only information about the linked list that we have is the pointer `start`. So we will traverse the list till the end to get the pointer `p` and then do the insertion. This is how we can obtain the pointer `p`.

```
p = start;
while(p->link!=NULL)
        p = p->link;
```

In traversal of list(3.1.1) our terminating condition was `(p!=NULL)`, because there we wanted the loop to terminate when `p` becomes `NULL`. Here we want the loop to terminate when `p` is pointing to the last node so the terminating condition is `(p->link!=NULL)`.

The following function `addatend()` inserts a node at the end of the list.

```
struct node *addatend(struct node *start,int data)
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        p = start;
        while(p->link!=NULL)
                p = p->link;
        p->link = tmp;
        tmp->link = NULL;
        return start;
}/*End of addatend()*/
```

3.1.3.4 Insertion in between the list nodes

We have to insert a node T between nodes P and Q.

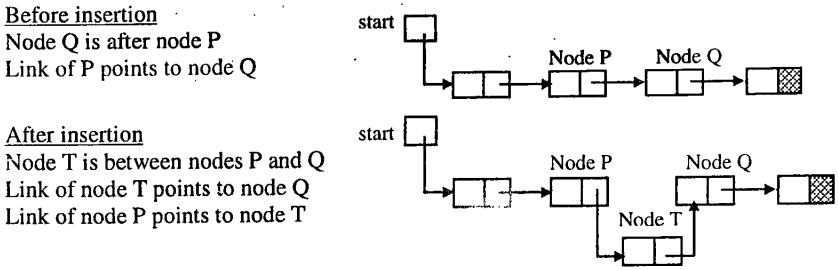


Figure 3.8 Insertion in between the list nodes

Suppose we have two pointers `p` and `q` pointing to nodes P and Q respectively. The two statements that should be written for insertion of node T are-

```
tmp->link = q;
p->link = tmp;
```

Before insertion address of node Q is in `p->link`, so instead of pointer `q` we can write `p->link`. Now the two statements for insertion can be written as-

```
tmp->link = p->link;
p->link = tmp;
```

Note : The order of these two statements is important, if you write them in the reverse order then you will lose your links. The address of node Q is in `p->link`, suppose we write the statement (`p->link = tmp;`) first then we will lose the address of node Q, there is no way to reach node Q and our list is broken. So first we should assign the address of node Q to the link of node T by writing (`tmp->link = p->link`). Now we have stored the address of node Q, so we are free to change `p->link`.

Now we will see three cases of insertion in between the nodes-

1. Insertion after a node
2. Insertion before a node
3. Insertion at a given position

The two statements of insertion(`tmp->link = p->link; p->link = tmp;`) will be written in all the three cases, but the way of finding the pointer `p` will be different.

3.1.3.4.1 Insertion after a node

In this case we are given a value from the list, and we have to insert the new node after the node that contains this value. Suppose the node P contains the given value, and node Q is its successor. We have to insert the new node T after node P, i.e. T is to be inserted between nodes P and Q. In the function given below, `data` is the new value to be inserted and `item` is the value contained in node P. For writing the two statements of insertion (`tmp->link = p->link; p->link = tmp;`), we need to find the pointer `p` which points to the node that contains `item`. The procedure is same as we have seen in searching of an element in the linked list.

```
struct node *addafter(struct node *start,int data,int item)
{
        struct node *tmp,*p;
        p = start;
        while(p!=NULL)
        {
                if(p->info == item)
                {
                        tmp = (struct node *)malloc(sizeof(struct node));
                        tmp->info = data;
                        tmp->link = p->link;
```

```
                    p->link = tmp;
                    return start;
                }
                p = p->link;
        }
        printf("%d not present in the list\n",item);
        return start;
}/*End of addafter()*/
```

Let us see what happens if `item` is present in the last node and we have to insert after the last node. In this case `p` points to last node and its link is `NULL`, so `tmp->link` is automatically assigned `NULL` and we don't have any need for a special case of insertion at the end. This function will work correctly even if we insert after the last node.

3.1.3.4.2 Insertion before a node

In this case we are given a value from the list, and we have to insert the new node before the node that contains this value. Suppose the node Q contains the given value, and node P is its predecessor. We have to insert the new node T before node Q, i.e. T is to be inserted between nodes P and Q. In the function given below, `data` is the new value to be inserted and `item` is the value contained in node Q. For writing the two statements of insertion (`tmp->link = p->link; p->link = tmp;`) we need to find the pointer `p` which points to the predecessor of the node that contains `item`. Since item is present in Q and we have to find pointer to node P, so here the condition for searching would be `if(p->link->info == item)` and terminating condition of the loop would be `(p->link! = NULL)`

```
struct node *addbefore(struct node *start,int data,int item)
{
        struct node *tmp,*p;
        if(start == NULL )
        {
                printf("List is empty\n");
                return start;
        }
        /*If data to be inserted before first node*/
        if(item == start->info)
        {
                tmp = (struct node *)malloc(sizeof(struct node));
                tmp->info = data;
                tmp->link = start;
                start = tmp;
                return start;
        }
        p = start;
        while(p->link!=NULL)
        {
                if(p->link->info == item)
                {
                        tmp = (struct node *)malloc(sizeof(struct node));
                        tmp->info = data;
                        tmp->link = p->link;
                        p->link = tmp;
                        return start;
                }
                p = p->link;
        }
        printf("%d not present in the list\n",item);
        return start;
}/*End of addbefore()*/
```

If the node is to be inserted before the first node, then that case has to be handled separately because we have to update `start` in this case. If the list is empty, start will be `NULL` and the term `start->info` will create problems so before checking the condition `if(item == start->info)` we should check for empty list.

3.1.3.4.3 Insertion at a given position

In this case we have to insert the new node at a given position. The two insertion statements are same as in the previous cases (`tmp->link = p->link; p->link = tmp;`).

The way of finding pointer `p` is different. If we have to insert at the first position we will have to update `start` so that case is handled separately.

```
struct node *addatpos(struct node *start,int data,int pos)
{
        struct node *tmp,*p;
        int i;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        if(pos==1)
        {
                tmp->link = start;
                start = tmp;
                return start;
        }
        p = start;
        for(i=1; i<pos-1 && p!=NULL; i++)
                p = p->link;
        if(p==NULL)
                printf("There are less than %d elements\n",pos);
        else
        {
                tmp->link = p->link;
                p->link = tmp;
        }
        return start;
}/*End of addatpos()*/
```

3.1.4 Creation of a Single Linked List

A list can be created using the insertion operations. First time we will have to insert into an empty list, and then we will keep on inserting nodes at the end of the list. The case of insertion in an empty list reduces to the case of insertion in the beginning, so for inserting the first node we will call `addatbeg()`, and then for insertion of all other nodes we will call `addatend()`.

```
struct node *create_list(struct node *start)
{
        int i,n,data;
        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        start = NULL;
        if(n==0)
                return start;
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        start = addatbeg(start,data);
        for(i=2; i<=n; i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                start = addatend(start,data);
        }
        return start;
```

```
}/*End of create_list()*/
```

3.1.5 Deletion in a Single Linked List

For deletion of any node, the pointers are rearranged so that this node is logically removed from the list. To physically remove the node and return the memory occupied by it to the pool of available memory we will use the function `free()`. We will take a pointer variable `tmp` which will point to the node being deleted so that after the pointers have been altered we will still have address of that node in `tmp` to free it. There can be four cases while deleting an element from a linked list.

1. Deletion of first node.
2. Deletion of the only node.
3. Deletion in between the list.
4. Deletion at the end.

In all the cases, at the end we should call `free(tmp)` to physically remove node T from the memory.

3.1.5.1 Deletion of first node

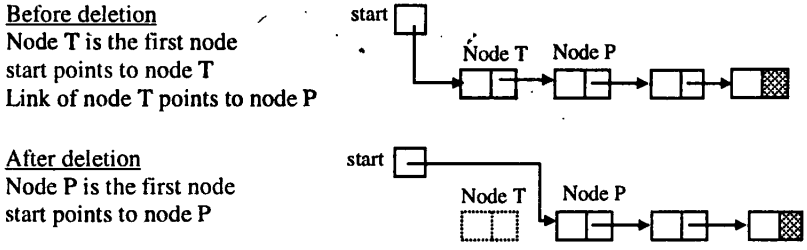


Figure 3.9 Deletion of first node

Since the node to be deleted is the first node, `tmp` will be assigned the address of first node.

```
tmp = start;
```

So now `tmp` points to the first node, which has to be deleted. Since `start` points to the first node of linked list, `start->link` will point to the second node of linked list. After deletion of first node, the second node(node P) would become the first one, so `start` should be assigned the address of the node P as-

```
start = start->link;
```

After this statement, `start` points to node P so now it is the first node of the list.

3.1.5.2 Deletion of the only node

If there is only one node in the list and we have to delete it, then after deletion the list would become empty and `start` would have `NULL` value.

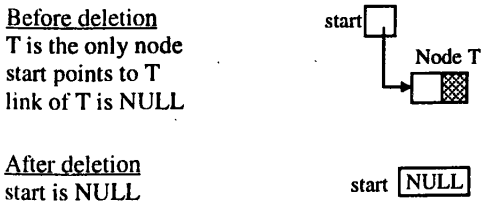


Figure 3.10 Deletion of the only node

```
tmp = start;
start = NULL;
```

In the second statement, we can write `start->link` instead of `NULL`. So this case reduces to the first one(3.1.5.1).

3.1.5.3 Deletion in between the list nodes

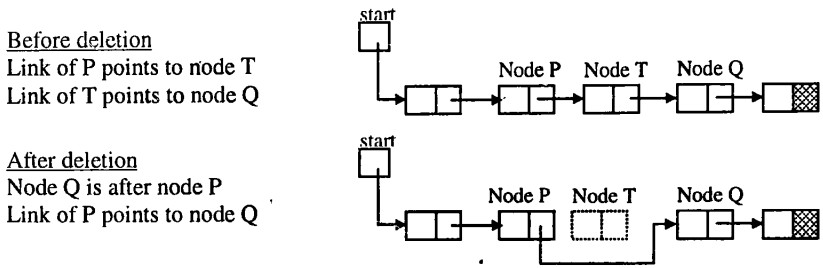


Figure 3.11 Deletion in between the list nodes

Suppose node T is to be deleted and pointer `tmp` points to it and we have pointers `p` and `q` which point to nodes P and Q respectively. For deletion of node T we will just link the predecessor of T(node P) to successor of T(node Q).

```
p->link = q;
```

The address of `q` is stored in `tmp->link` so instead of `q` we can write `tmp->link` in the above statement.

```
p->link = tmp->link;
```

The value to be deleted is in node T and we need a pointer to its predecessor which is node P, so as in `addbefore()` our condition for searching will be `if(p->link->info == data)`. Here `data` is the element to be deleted.

```
p = start;
while(p->link!= NULL)
{
        if(p->link->info == data)
        {
                tmp = p->link;
                p->link = tmp->link;
                free(tmp);
                return start;
        }
        p = p->link;
}
```

3.1.5.4 Deletion at the end of the list

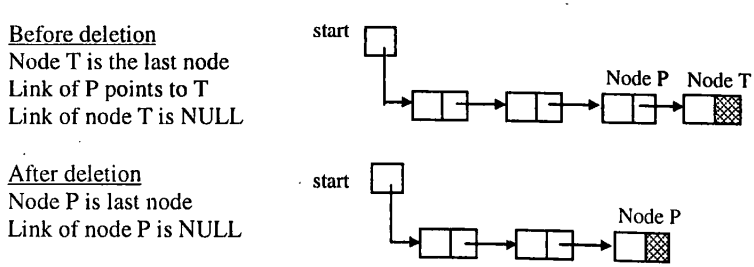


Figure 3.12 Deletion at the end of the list

If `p` is a pointer to node P, then node T can be deleted by writing the following statement.

```
p->link = NULL;
```

Since link of node T is NULL, in the above statement instead of NULL we can write tmp->link. Hence we can write this statement as-

```
p->link = tmp->link;
```

So we can see that this case reduces to the previous case(3.1.5.3).

Now we will write a function for deletion of an element from the list. We can't delete from an empty list so firstly we will check if the list is empty. If the element to be deleted is the first element of the list then it is deleted accordingly and we return from the function.

Now the element can either be in between the list nodes or at end of the list. These two cases can be handled in the same way. If we reach the end of list and node containing the value is not found, then a message is displayed.

```
struct node *del(struct node *start,int data)
{
        struct node *tmp,*p;
        if(start == NULL)
        {
                printf("List is empty\n");
                return start;
        }
        if(start->info == data)  /*Deletion of first node*/
        {
                tmp = start;
                start = start->link;
                free(tmp);
                return start;
        }
        p = start;   /*Deletion in between or at the end*/
        while(p->link!=NULL)
        {
                if(p->link->info == data)
                {
                        tmp = p->link;
                        p->link = tmp->link;
                        free(tmp);
                        return start;
                }
                p = p->link;
        }
        printf("Element %d not found\n",data);
        return start;
}/*End of del()*/
```

3.1.6 Reversing a Single Linked List

The following changes need to be done in a single linked list for reversing it.

1. First node should become the last node of linked list.
2. Last node should become the first node of linked list and now start should point to it.
3. Link of 2nd node should point to 1st node, link of 3rd node should point to 2nd node and so on.

Let us take a single linked list and reverse it.

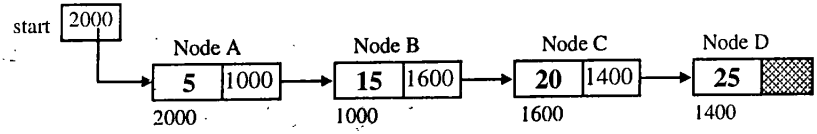


Figure 3.13

(i) Node A is the first node so `start` points to it.
(ii) Node D is the last node so its link is NULL.
(iii) Link of A points to B, link of B points to C and link of C points to D.

After reversing, the linked list would be-

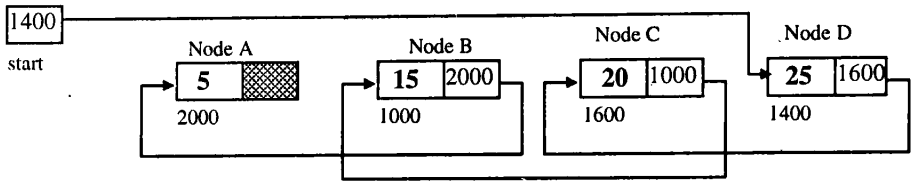


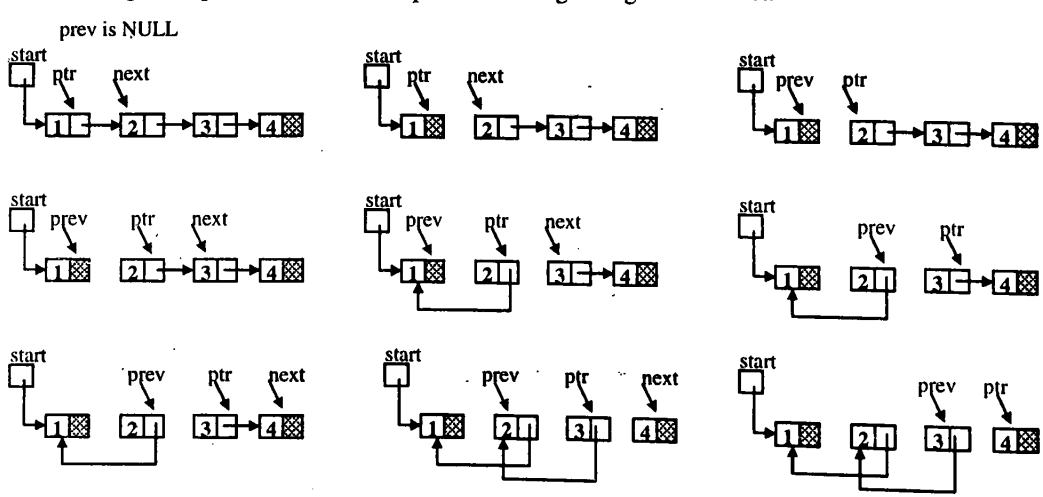
Figure 3.14

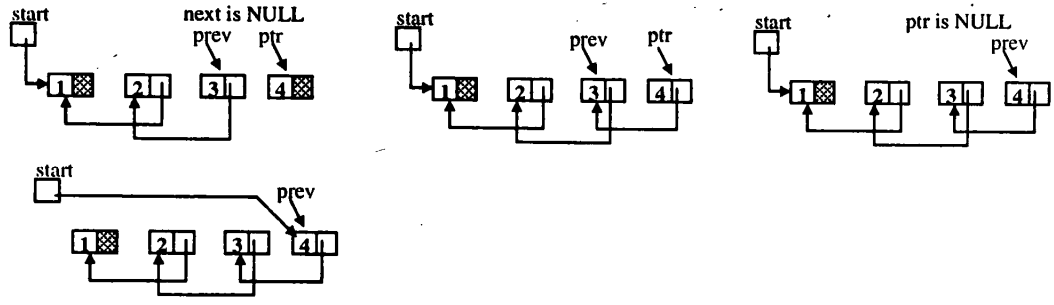
(i) Node D is the first node so `start` points to it.
(ii) Node A is the last node so its link is NULL.
(iii) Link of D points to C, link of C points to B and link of B points to A.

Now let us see how we can make a function for the reversal of a linked list. We will take three pointers `prev`, `ptr` and `next`. Initially the pointer `ptr` will point to `start` and `prev` will be NULL. In each pass, first the link of pointer `ptr` is stored in pointer `next` and after that the link of `ptr` is changed so that it points to its previous node. The pointers `prev` and `ptr` are moved forward. Here is the function `reverse()` that reverses a linked list.

```
struct node *reverse(struct node *start)
{
        struct node *prev, *ptr, *next;
        prev = NULL;
        ptr = start;
        while(ptr!=NULL)
        {
                next = ptr->link;
                ptr->link = prev;
                prev = ptr;
                ptr = next;
        }
        start = prev;
        return start;
}/*End of reverse()*/
```

The following example shows all the steps of reversing a single linked list.





3.2 Doubly linked list

The linked list that we have studied contained only one link, this is why these lists are called single linked lists or one way lists. We could move only in one direction because each node has address of next node only. Suppose we are in the middle of linked list and we want the address of previous node then we have no way of doing this except repeating the traversal from the starting node. To overcome this drawback of single linked list we have another data structure called doubly linked list or two way list, in which each node has two pointers. One of these pointers points to the next node and the other points to the previous node. The structure for a node of doubly linked list can be declared as-

```
struct node{
                struct node *prev;
                int info;
                struct node *next;
        };
```

Here `prev` is a pointer that will contain the address of previous node and `next` will contain the address of next node in the list. So we can move in both directions at any time. The `next` pointer of last node and `prev` pointer of first node are NULL.

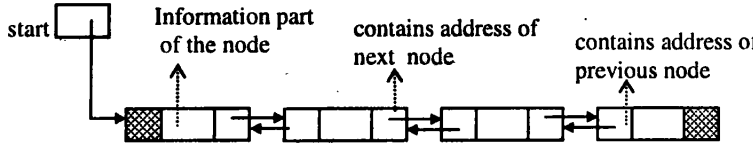


Figure 3.15 Doubly linked list

The basic logic for all operations is same as in single linked list, but here we have to do a little extra work because there is one more pointer that has to be updated each time. The `main()` function for the program of doubly linked list is-

```
/*P3.2 Program of doubly linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        struct node *prev;
        int info;
        struct node *next;
};
struct node *create_list(struct node *start);
void display(struct node *start);
struct node *addtoempty(struct node *start,int data);
struct node *addatbeg(struct node *start,int data);
struct node *addatend(struct node *start,int data);
struct node *addafter(struct node *start,int data,int item);
struct node *addbefore(struct node *start,int data,int item );
```

```
struct node *del(struct node *start,int data);
struct node *reverse(struct node *start);

main()
{
        int choice,data,item;
        struct node *start=NULL;
        while(1)
        {
                printf("1.Create List\n");
                printf("2.Display\n");
                printf("3.Add to empty list\n");
                printf("4.Add at beginning\n");
                printf("5.Add at end\n");
                printf("6.Add after\n");
                printf("7.Add before\n");
                printf("8.Delete\n");
                printf("9.Reverse\n");
                printf("10.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        start=create_list(start);
                        break;
                 case 2:
                        display(start);
                        break;
                 case 3:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start=addtoempty(start,data);
                        break;
                 case 4:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start=addatbeg(start,data);
                        break;
                 case 5:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start=addatend(start,data);
                        break;
                 case 6:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        printf("Enter the element after which to insert : ");
                        scanf("%d",&item);
                        start=addafter(start,data,item);
                        break;
                 case 7:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        printf("Enter the element before which to insert : ");
                        scanf("%d",&item);
                        start=addbefore(start,data,item);
                        break;
                 case 8:
                        printf("Enter the element to be deleted : ");
                        scanf("%d",&data);
                        start=del(start,data);
                        break;
                 case 9:
```

```
                    start=reverse(start);
                    break;
              case 10:
                    exit(1);
              default:
                    printf("Wrong choice\n");
        }/*End of switch*/
    }/*End of while*/
}/*End of main ()*/
```

3.2.1 Traversing a doubly linked List

The function for traversal of doubly linked list is similar to that of single linked list.

```
void display(struct node *start)
{
        struct node *p;
        if(start==NULL)
        {
                printf("List is empty\n");
                return;
        }
        p = start;
        printf("List is :\n");
        while(p!=NULL)
        {
                printf("%d ",p->info);
                p = p->next;
        }
        printf("\n");
}/*End of display()*/
```

3.2.2 Insertion in a doubly linked List

We will study all the four cases of insertion in a doubly linked list.

1. Insertion at the beginning of the list.
2. Insertion in an empty list.
3. Insertion at the end of the list.
4. Insertion in between the nodes

3.2.2.1 Insertion at the beginning of the list

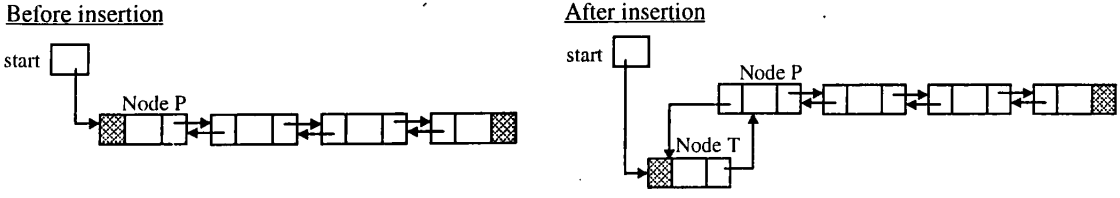


Figure 3.16 Insertion at the beginning of the list

Node T has become the first node so its `prev` should be `NULL`.

```
tmp->prev = NULL;
```

The `next` part of node T should point to node P, and address of node P is in `start` so we should write-

```
tmp->next = start;
```

Node T is inserted before node P so `prev` part of node P should now point to node T.

```
start->prev = tmp;
```

Now node T has become the first node so `start` should point to it.

```
start = tmp;
```

```
struct node *addatbeg(struct node *start,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        tmp->prev = NULL;
        tmp->next = start;
        start->prev = tmp;
        start = tmp;
        return start;
}/*End of addatbeg()*/
```

3.2.2.2 Insertion in an empty list

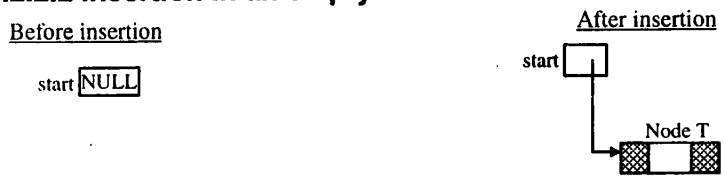


Figure 3.17 Insertion in an empty list

Node T is the first node so its `prev` part should be `NULL`, and it is also the last node so its `next` part should also be `NULL`. Node T is the first node so `start` should point to it.

```
tmp->prev = NULL;
tmp->next = NULL;
start = tmp;
```

In single linked list this case had reduced to the case of insertion at the beginning but here it is not so.

```
struct node *addtoempty(struct node *start,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        tmp->prev = NULL;
        tmp->next = NULL;
        start=tmp;
        return start;
}/*End of addtoempty()*/
```

3.2.2.3 Insertion at the end of the list

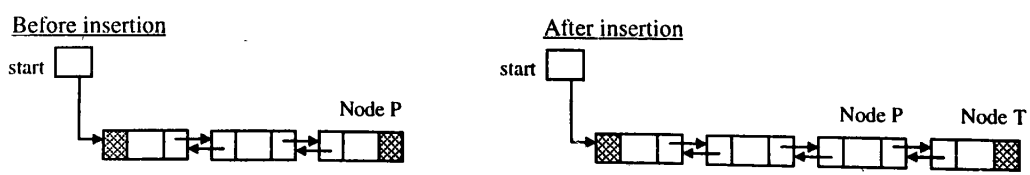


Figure 3.18 Insertion at the end of the list

Suppose `p` is a pointer pointing to the node P which is the last node of the list.

Node T becomes the last node so its `next` should be `NULL`

```
tmp->next = NULL;
```

`next` part of node P should point to node T

```
p->next = tmp;
```

`prev` part of node T should point to node P

```
tmp->prev = p;
```

```
struct node *addatend(struct node *start,int data)
{
```

```
        struct node *tmp,*p;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        p = start;
        while(p->next!=NULL)
                p = p->next;
        p->next = tmp;
        tmp->next = NULL;
        tmp->prev = p;
        return start;
}/*End of addatend()*/
```

3.2.2.4 Insertion in between the nodes

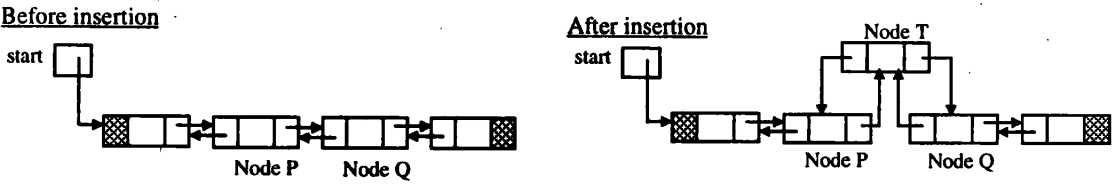


Figure 3.19 Insertion in between the nodes

Suppose pointers p and q point to nodes P and Q respectively.
Node P is before node T so prev of node T should point to node P
`tmp->prev = p;`
Node Q is after node T so next part of node T should point to node Q
`tmp->next = q;`
Node T is before node Q so prev part of node Q should point to node T
`q->prev = tmp;`
Node T is after node P so next part of node P should point to node T
`p->next = tmp;`

Now we will see how we can write the function `addafter()` for doubly linked list. We are given a value and the new node is to be inserted after the node containing this value.
Suppose node P contains this value so we have to add new node after node P. As in single linked list here also we can traverse the list and find a pointer p pointing to node P. Now in the four insertion statements we can replace q by `p->next`.

```
tmp->prev = p;      ->      tmp->prev = p;
tmp->next = q;      ->      tmp->next = p->next;
q->prev = tmp;      ->      p->next->prev = tmp;
p->next = tmp;      ->      p->next = tmp;
```

Note that `p->next` should be changed at the end because we are using it in previous statements.
In single linked list we had seen that the case of inserting after the last node was handled automatically. But here when we insert after the last node the third statement `(p->next->prev = tmp;)` will create problems. The pointer p points to last node so its next is NULL hence the term `p->next->prev` is meaningless here. To avoid this problem we can put a check like this-

```
if(p->next!=NULL)
        p->next->prev = tmp;

struct node *addafter(struct node *start,int data,int item)
{
        struct node *tmp,*p;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        p = start;
        while(p!=NULL,
```

```
        {
                if(p->info == item)
                {
                        tmp->prev = p;
                        tmp->next = p->next;
                        if(p->next!=NULL)
                                p->next->prev = tmp;
                        p->next = tmp;
                        return start;
                }
                p = p->next;
        }
        printf("%d not present in the list\n",item);
        return start;
}/*End of addafter()*/
```

Now we will see how to write function `addbefore()` for doubly linked list. In this case suppose we have to insert the new node before node Q, so we will traverse the list and find a pointer `q` to node Q. In single linked list we had to find the pointer to predecessor but here there is no need to do so because we can get the address of predecessor by `q->prev`. So just replace `p` by `q->prev` in the four insertion statements.

```
tmp->prev = p;      ->      tmp->prev = q->prev;
tmp->next = q;      ->      tmp->next = q;
q->prev = tmp;      ->      q->prev = tmp;
p->next = tmp;      ->      q->prev->next = tmp;
```

`q->prev` should be changed at the end because it being used in other statements. Thus third statement should be the last one.

```
tmp->prev = q->prev;
tmp->next = q;
q->prev->next = tmp;
q->prev = tmp;
```

As in single linked list, here also we will have to handle the case of insertion before the first node separately.

```
struct node *addbefore(struct node *start,int data,int item)
{
        struct node *tmp,*q;
        if(start==NULL)
        {
                printf("List is empty\n");
                return start;
        }
        if(start->info == item)
        {
                tmp = (struct node *)malloc(sizeof(struct node));
                tmp->info = data;
                tmp->prev = NULL;
                tmp->next = start;
                start->prev = tmp;
                start = tmp;
                return start;
        }
        q = start;
        while(q!=NULL)
        {
                if(q->info == item)
                {
                        tmp = (struct node *)malloc(sizeof(struct node));
                        tmp->info = data;
                        tmp->prev = q->prev;
                        tmp->next = q;
                        q->prev->next = tmp;
                        q->prev = tmp;
```

```
                    return start;
                }
                q = q->next;
        }
        printf("%d not present in the list\n",item);
        return start;
}/*End of addbefore()*/
```

3.2.3 Creation of List

For inserting the first node we will call `addtoempty()`, and then for insertion of all other nodes we will call `addatend()`.

```
struct node *create_list(struct node *start)
{
        int i,n,data;
        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        start=NULL;
        if(n==0)
                return start;
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        start=addtoempty(start,data);
        for(i=2;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                start=addatend(start,data);
        }
        return start;
}/*End of create_list()*/
```

3.2.4 Deletion from doubly linked list

As in single linked list, here also the node is first logically removed by rearranging the pointers and then it is physically removed by calling the function `free()`. Let us study the four cases of deletion-.

1. Deletion of first node.
2. Deletion of the only node.
3. Deletion in between the nodes.
4. Deletion at the end.

In all the cases we will take a pointer variable `tmp` which will point to the node being deleted.

3.2.4.1 Deletion of the first node

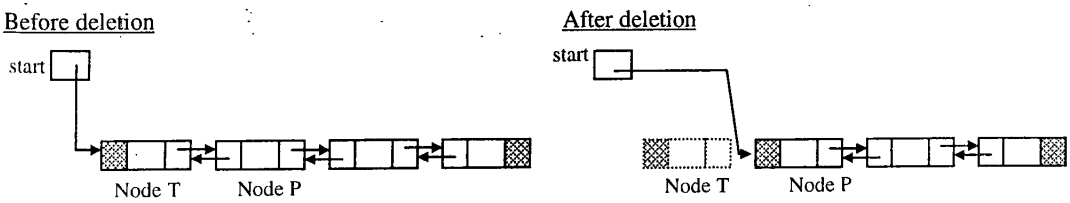


Figure 3.20 Deletion of the first node

`tmp` will be assigned the address of first node.

```
tmp = start;
```

`start` will be updated so that now it points to node P

```
start = start->next;
```

Now node P is the first node so its `prev` part should contain `NULL`.

```
start->prev = NULL
```

3.2.4.2 Deletion of the only node

Figure 3.21 Deletion of the only node

The two statements for deletion will be -

```
tmp = start;
start = NULL;
```

In single linked list we had seen that this case reduced to the previous one. Let us see what happens here. We can write `start->next` instead of `NULL` in the second statement, but then also this case does not reduce to the previous one. This is because of the third statement in the previous case, since start becomes `NULL`, the term `start->prev` is meaningless.

3.2.4.3 Deletion in between the nodes

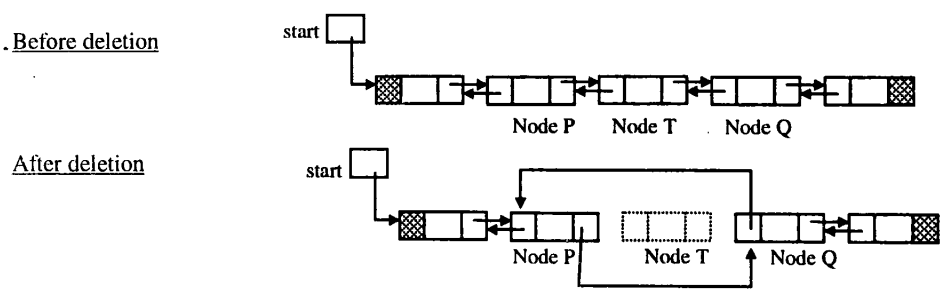


Figure 3.22 Deletion in between the nodes

Suppose we have to delete node T, and let pointers `p`, `tmp` and `q` point to nodes P, T and Q respectively. The two statements for deleting node T can be written as-

```
p->next = q;
q->prev = p;
```

The address of `q` is in `tmp->next` so we can replace `q` by `tmp->next`.

```
p->next = tmp->next;
tmp->next->prev = p;
```

The address of `p` is stored in `tmp->prev` so we can replace `p` by `tmp->prev`.

```
tmp->prev->next = tmp->next;
tmp->next->prev = tmp->prev;
```

So we need only pointer to a node for deleting it.

3.2.4.4 Deletion at the end of the list

Suppose node T is to be deleted and pointers `tmp` and `p` point to nodes T and P respectively. The deletion can be performed by writing the following statement.

```
p->next = NULL;
```

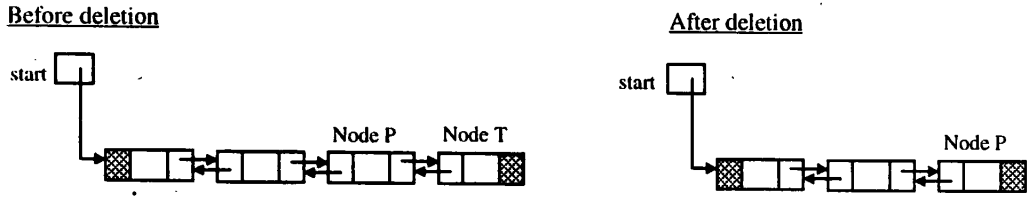


Figure 3.23 Deletion at the end of the list

The address of node P is stored in `tmp->prev`, so we can replace `p` by `tmp->prev`.

```
tmp->prev->next = NULL;
```

In single linked list, this case reduced to the previous case but here it won't.

```
struct node *del(struct node *start,int data)
{
        struct node *tmp;
        if(start == NULL)
        {
                printf("List is empty\n");
                return start;
        }
        if(start->next == NULL) /*Deletion of only node*/
                if(start->info == data)
                {
                        tmp = start;
                        start = NULL;
                        free(tmp);
                        return start;
                }
                else
                {
                        printf("Element %d not found\n",data);
                        return start;
                }
        if(start->info == data) /*Deletion of first node*/
        {
                tmp = start;
                start = start->next;
                start->prev = NULL;
                free(tmp);
                return start;
        }
        tmp = start->next; /*Deletion in between*/
        while(tmp->next!=NULL )
        {
                if(tmp->info == data)
                {
                        tmp->prev->next = tmp->next;
                        tmp->next->prev = tmp->prev;
                        free(tmp);
                        return start;
                }
                tmp = tmp->next;
        }
        if(tmp->info == data) /*Deletion of last node*/
        {
                tmp->prev->next = NULL;
                free(tmp);
                return start;
        }
        printf("Element %d not found\n",data);
        return start;
}/*End of del()*/
```

3.2.5 Reversing a doubly linked list

Let us take a doubly linked list and see what changes need to be done for its reversal. The following figure shows a double linked list and the reversed linked list.

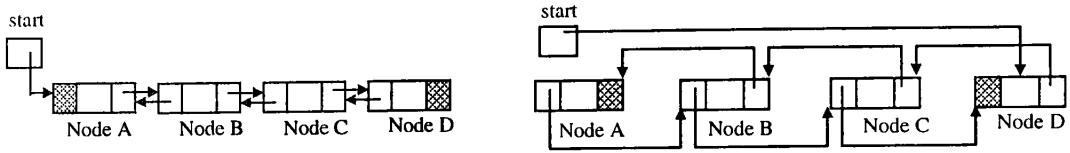


Figure 3.24

In the reversed list-
(i) `start` points to Node D.
(ii) Node D is the first node so its `prev` is `NULL`.
(iii) Node A is the last node so its `next` is `NULL`.
(iv) `next` of D points to C, `next` of C points to B and `next` of B points to A.
(v) `prev` of A points to B, `prev` of B points to C, `prev` of C points to D.

For making the function of reversal of doubly linked list we will need only two pointers.

```
struct node *reverse(struct node *start)
{
        struct node *p1,*p2;
        p1 = start;
        p2 = p1->next;
        p1->next = NULL;
        p1->prev=p2;
        while(p2!=NULL)
        {
                p2->prev = p2->next;
                p2->next = p1;
                p1 = p2;
                p2 = p2->prev;
        }
        start = p1;
        printf("List reversed\n");
        return start;
}/*End of reverse()*/
```

In a doubly linked list we have an extra pointer which consumes extra space, and maintenance of this pointer makes operations lengthy and time consuming. So doubly linked lists are beneficial only when we frequently need the predecessor of a node.

3.3 Circular linked list

In a single linked list, for accessing any node of linked list, we start traversing from first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of single linked list. In a single linked list, link part of last node is `NULL`, if we utilize this link to point to the first node then we can have some advantages. The structure thus formed is called a circular linked list. The following figure shows a circular linked list.

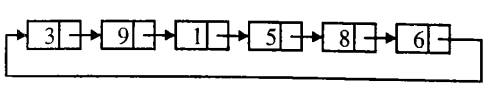


Figure 3.25

Each node has a successor and all the nodes form a ring. Now we can access any node of the linked list without going back and starting traversal again from first node because list is in the form of a circle and we can go from last node to first node.

We take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then `last->link` will point to the first node.

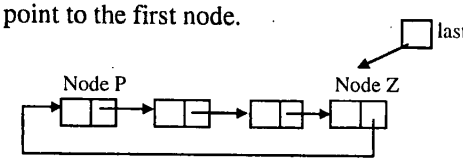


Figure 3.26

In the figure 3.26, the pointer `last` points to node Z and `last->link` points to node P. Let us see why we have taken a pointer that points to the last node instead of first node. Suppose we take a pointer `start` pointing to first node of circular linked list. Take the case of insertion of a node in the beginning.

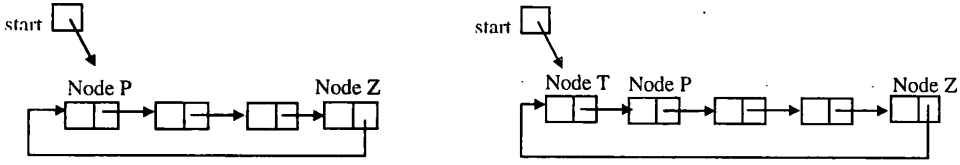


Figure 3.27

For insertion of node T in the beginning we need the address of node Z, because we have to change the link of node Z and make it point to node T. So we will have to traverse the whole list. For insertion at the end it is obvious that the whole list has to be traversed. If instead of pointer `start` we take a pointer to the last node then in both the cases there won't be any need to traverse the whole list. So insertion in the beginning or at the end takes constant time irrespective of the length of the list.

If the circular list is empty the pointer last is NULL, and if the list contains only one element then the link of `last` points to `last`.

Now let us see different operations on the circular linked list. The algorithms are similar to that of single linked list but we have to make sure that after completing any operation the link of last node points to the first.

```
/*P3.3 Program of circular linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
struct node *create_list(struct node *last);
void display(struct node *last);
struct node *addtoempty(struct node *last,int data);
struct node *addatbeg(struct node *last,int data);
struct node *addatend(struct node *last,int data);
struct node *addafter(struct node *last,int data,int item);
struct node *del(struct node *last,int data);

main()
{
        int choice,data,item;
        struct node *last=NULL;

        while(1)
        {
                printf("1.Create List\n");
                printf("2.Display\n");
                printf("3.Add to empty list\n");
                printf("4.Add at beginning\n");
                printf("5.Add at end\n");
                printf("6.Add after \n");
                printf("7.Delete\n");
                printf("8.Quit\n");
```

```
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1:
                        last=create_list(last);
                        break;
                 case 2:
                        display(last);
                        break;
                 case 3:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        last=addtoempty(last,data);
                        break;
                 case 4:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        last=addatbeg(last,data);
                        break;
                 case 5:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        last=addatend(last,data);
                        break;
                 case 6:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        printf("Enter the element after which to insert : ");
                        scanf("%d",&item);
                        last=addafter(last,data,item);
                        break;
                 case 7:
                        printf("Enter the element to be deleted : ");
                        scanf("%d",&data);
                        last=del(last,data);
                        break;
                 case 8:
                        exit(1);
                 default:
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
```

3.3.1 Traversal in circular linked list

First of all we will check if the list is empty. After that we will take a pointer p and make it point to the first node.

```
p = last->link;
```

The link of last node does not contain NULL but contains the address of first node so here the terminating condition of our loop becomes (p!=last->link) .We have used a do-while loop in the function display() because if we take a while loop then the terminating condition will be satisfied in the first time only and the loop will not execute at all.

```
void display(struct node *last)
{
        struct node *p;

        if(last == NULL)
        {
                printf("List is empty\n");
```

```
                return;
        }
        p = last->link;
        do
        {
                printf("%d ",p->info);
                p = p->link;
        }while(p!=last->link);
        printf("\n");
}/*End of display()*/
```

3.3.2 Insertion in a circular Linked List

3.3.2.1 Insertion at the beginning of the list

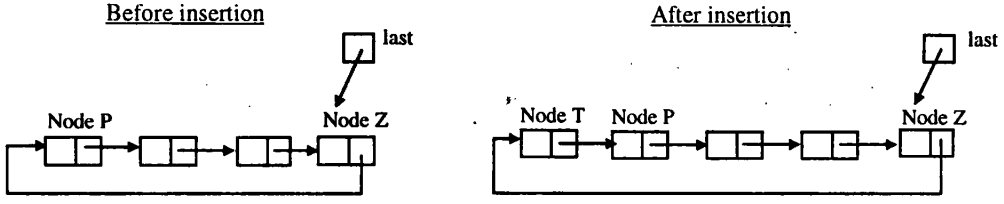


Figure 3.27 Insertion at the beginning of the list

Before insertion, P is the first node so `last->link` points to node P.

After insertion, link of node T should point to node P and address of node P is in `last->link`

```
tmp->link = last->link;
```

Link of last node should point to node T.

```
last->link = tmp;
```

```
struct node *addatbeg(struct node *last,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        tmp->link = last->link;
        last->link = tmp;
        return last;
}/*End of addatbeg()*/
```

3.3.2.2 Insertion in an empty list

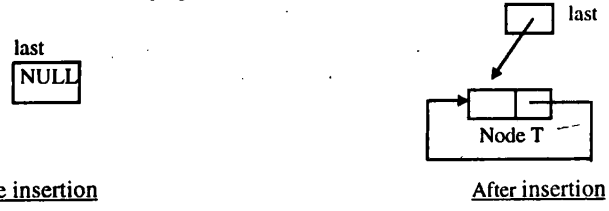


Figure 3.29 Insertion in an empty list

After insertion, T is the last node so pointer `last` points to node T.

```
last = tmp;
```

We know that `last->link` always points to the first node, here T is the first node so `last->link` points to node T (or `last`).

```
last->link = last;
```

```
struct node *addtoempty(struct node *last,int data)
```

```
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        last = tmp;
        last->link = last;
        return last;
}/*End of addtoempty()*/
```

3.3.2.3 Insertion at the end of the list

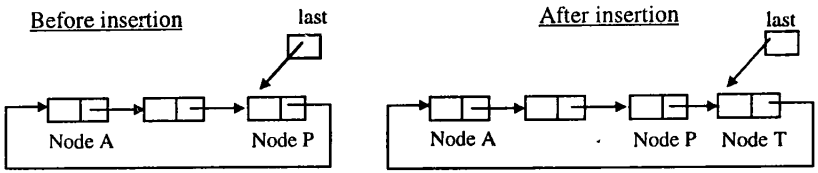


Figure 3.30 Insertion at the end of the list

Before insertion, `last` points to node P and `last->link` points to node A

Link of T should point to node A and address of node A is in `last->link`.

```
tmp->link = last->link;
```

Link of node P should point to node T

```
last->link = tmp;
```

`last` should point to node T

```
last = tmp;
```

The order of the above three statements is important.

```
struct node *addatend(struct node *last,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        tmp->link = last->link;
        last->link = tmp;
        last = tmp;
        return last;
}/*End of addatend()*/
```

3.3.2.4 Insertion in between the nodes

The logic for insertion in between the nodes is same as in single linked list. If insertion is done after the last node then the pointer `last` should be updated. The function `addafter()` is given below-

```
struct node *addafter(struct node *last,int data,int item)
{
        struct node *tmp,*p;
        p = last->link;
        do
        {
                if(p->info == item)
                {
                        tmp = (struct node *)malloc(sizeof(struct node));
                        tmp->info = data;
                        tmp->link = p->link;
                        p->link = tmp;
                        if(p==last)
                                last = tmp;
```

```
                        return last;
                }
                p = p->link;
        }while(p!=last->link);
        printf("%d not present in the list\n",item);
        return last;
}/*End of addafter()*/
```

3.3.3 Creation of circular linked list

For inserting the first node we will call `addtoempty()`, and then for insertion of all other nodes we will call `addatend()`.

```
struct node *create_list(struct node *last)
{
        int i,n,data;
        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        last=NULL;
        if(n==0)
                return last;
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        last=addtoempty(last,data);
        for(i=2; i<=n; i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                last=addatend(last,data);
        }
        return last;
}/*End of create_list()*/
```

3.3.4 Deletion in circular linked list

3.3.4.1 Deletion of the first node

Before deletion, `last` points to node Z and `last->link` points to node T

After deletion, link of node Z should point to node A, so `last->link` should point to node A. Address of node A is in link of node T.

```
last->link = tmp->link;
```

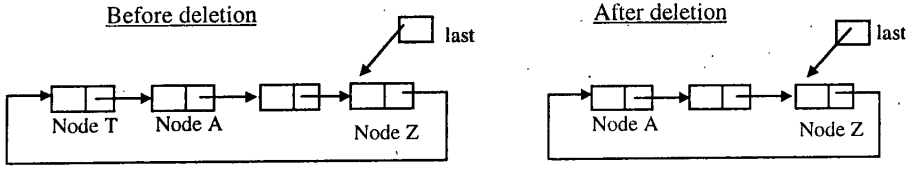


Figure 3.31 Deletion of the first node

3.3.4.2 Deletion of the only node

If there is only one element in the list then we assign `NULL` value to `last` pointer because after deletion there will be no node in the list.

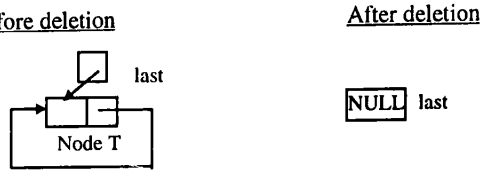


Figure 3.32 Deletion of the only node

There will be only one node in the list if link of last node points to itself. After deletion the list will become empty so NULL is assigned to last.

```
last = NULL;
```

3.3.4.3 Deletion in between the nodes

Deletion in between is same as in single linked list

3.3.4.4 Deletion at the end of the list

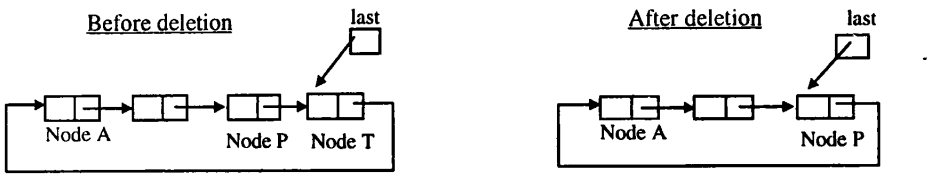


Figure 3.33 Deletion at the end of the list

Before deletion, `last` points to node T and `last->link` points to node A, `p` is a pointer to node P. After deletion, link of node P should point to node A. Address of node A is in `last->link`.

```
p->link = last->link;
```

Now P is the last node so `last` should point to node P.

```
last = p;
```

```
struct node *del(struct node *last,int data)
{
        struct node *tmp,*p;
        if(last == NULL)
        {
                printf("List is empty\n");
                return last;
        }
        if(last->link == last && last->info == data) /*Deletion of only node*/
        {
                tmp = last;
                last = NULL;
                free(tmp);
                return last;
        }
        if(last->link->info == data)  /*Deletion of first node*/
        {
                tmp = last->link;
                last->link = tmp->link;
                free(tmp);
                return last;
        }
        p = last->link;         /*Deletion in between*/
        while(p->link!=last)
        {
```

```
            if(p->link->info == data)
            {
                    tmp = p->link;
                    p->link = tmp->link;
                    free(tmp);
                    return last;
            }
            p = p->link;
        }
        if(last->info == data) /*Deletion of last node*/
        {
                tmp = last;
                p->link = last->link;
                last = p;
                free(tmp);
                return last;
        }
        printf("Element %d not found\n",data);
        return last;
}/*End of del()*/
```

We have studied circular lists which are singly linked. Double linked lists can also be made circular. In this case the next pointer of last node points to first node, and the `prev` pointer of first node points to the last node.

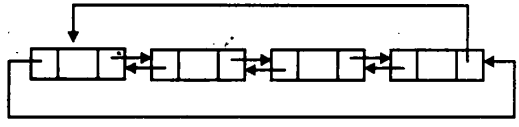


Figure 3.34 Circular double linked list

3.4 Linked List with Header Node

Header node is a dummy node that is present at the beginning of the list and its link part is used to store the address of the first actual node of the list. The info part of this node may be empty or can be used to contain useful information about the list like count of elements currently present in the list. The figure 3.35(a) shows a single linked list with 4 actual nodes and a header node, and the figure (b) shows an empty list with a header node.

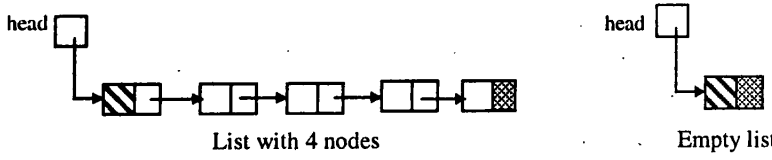


Figure 3.35 Single linked list with header node

The pointer `head` points to the header node and `head->link` gives the address of first true node of the list. If there is no node in the list then `head->link` will be `NULL`.

The header node is never deleted; it exists even if the list is empty. So it may be declared while writing the program instead of dynamically allocating memory for it.

The use of header nodes makes the program simple and faster. Since the list is never empty because header node is always there, we can avoid some special cases of empty list and that of insertion and deletion in the beginning. For example in the function `del()` and `addbefore()` that we had written for single linked list, we can drop the first two cases if we take a header node in our list.

The following program performs operations on a single linked list with header node. The logic of all operations is similar to that of single linked list without header but some cases have been removed.

```
/*P3.4 Program of single linked list with header node*/
#include<stdio.h>
#include<stdlib.h>
```

```
struct node
{
        int info;
        struct node *link;
};
struct node *create_list(struct node *head);
void display(struct node *head);
struct node *addatend(struct node *head,int data);
struct node *addbefore(struct node *head,int data,int item );
struct node *addatpos(struct node *head,int data,int pos);
struct node *del(struct node *head,int data);
struct node *reverse(struct node *head);

main()
{
        int choice,data,item,pos;
        struct node *head;
        head = (struct node *)malloc(sizeof(struct node));
        head->info = 0;
        head->link = NULL;
        head = create_list(head);
        while(1)
        {
                printf("1.Display\n");
                printf("2.Add at end\n");
                printf("3.Add before node\n");
                printf("4.Add at position\n");
                printf("5.Delete\n");
                printf("6.Reverse\n");
                printf("7.Quit\n\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        display(head);
                        break;
                 case 2:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        head = addatend(head,data);
                        break;
                 case 3:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        printf("Enter the element before which to insert : ");
                        scanf("%d",&item);
                        head = addbefore(head,data,item);
                        break;
                 case 4:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        printf("Enter the position at which to insert : ");
                        scanf("%d",&pos);
                        head = addatpos(head,data,pos);
                        break;
                 case 5:
                        printf("Enter the element to be deleted : ");
                        scanf("%d",&data);
                        head = del(head,data);
                        break;
                 case 6:
                        head = reverse(head);
                        break;
```

```
                case 7:
                        exit(1);
                default:
                        printf("Wrong choice\n\n");
                }/*End of switch */
        }/*End of while */
}/*End of main()*/

struct node *create_list(struct node *head)
{
        int i,n,data;
        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                head = addatend(head,data);
        }
        return head;
}/*End of create_list()*/

void display(struct node *head)
{
        struct node *p;
        if(head->link==NULL)
        {
                printf("List is empty\n");
                return;
        }
        p = head->link;
        printf("List is :\n");
        while(p!=NULL)
        {
                printf("%d ",p->info);
                p=p->link;
        }
        printf("\n");
}/*End of display() */

struct node *addatend(struct node *head,int data)
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        p = head;
        while(p->link!=NULL)
                p = p->link;
        p->link = tmp;
        tmp->link = NULL;
        return head;
}/*End of addatend()*/

struct node *addbefore(struct node *head,int data,int item)
{
        struct node *tmp,*p;
        p = head;
        while(p->link!=NULL)
        {
                if(p->link->info==item)
                {
                        tmp = (struct node *)malloc(sizeof(struct node));
                        tmp->info = data;
                        tmp->link = p->link;
                        p->link = tmp;
                        return head;
```

```
                }
                p = p->link;
        }
        printf("%d not present in the list\n",item);
        return head;
}/*End of addbefore()*/

struct node *addatpos(struct node *head,int data,int pos)
{
        struct node *tmp,*p;
        int i;
        tmp = (struct node *)malloc(sizeof(struct node) );
        tmp->info = data;
        p = head;
        for(i=1;i<=pos-1;i++)
        {
                p = p->link;
                if(p==NULL)
                {
                        printf("There are less than %d elements\n",pos);
                        return head;
                }
        }
        tmp->link = p->link;
        p->link = tmp;
        return head;
}/*End of addatpos()*/

struct node *del(struct node *head, int data)
{
        struct node *tmp,*p;
        p = head;
        while(p->link!=NULL)
        {
                if(p->link->info==data)
                {
                        tmp = p->link;
                        p->link = tmp->link;
                        free(tmp);
                        return head;
                }
                p = p->link;
        }
        printf("Element %d not found\n",data);
        return head;
}/*End of del()*/

struct node *reverse(struct node *head)
{
        struct node *prev, *ptr, *next;
        prev = NULL;
        ptr = head->link;
        while(ptr!=NULL)
        {
                next = ptr->link;
                ptr->link = prev;
                prev = ptr;
                ptr = next;
        }
        head->link = prev;
        return head;
}
```

The header node can be attached to circular linked lists and doubly linked lists also. The following figure shows circular single linked list with header node.

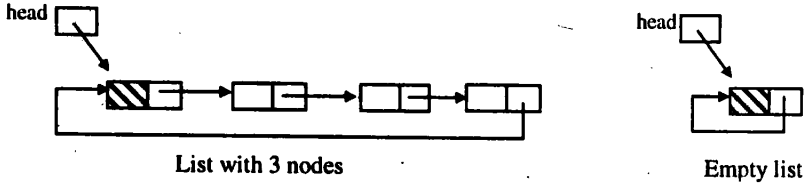
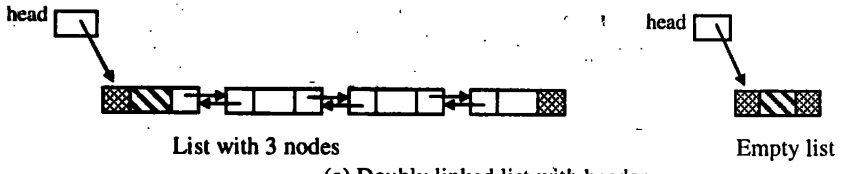


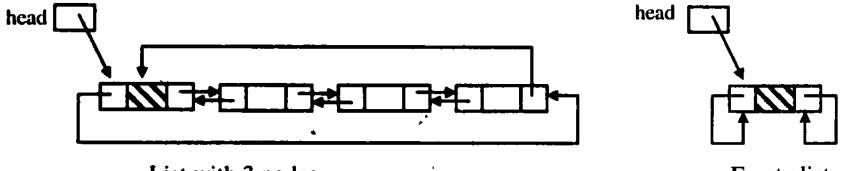
Figure 3.36 Circular single linked list with header

Here the external pointer points to the header node rather than at the end.

The following figures show doubly linked list and doubly linked circular list with header nodes.



(a) Doubly linked list with header



(b) Doubly linked circular list with header

Figure 3.37

3.5 Sorted linked list

In some applications, it is better if the elements in the list are kept in sorted order. For maintaining a list in sorted order we have to insert the nodes in proper place. Let us take an ascending order linked list and insert some elements in it.

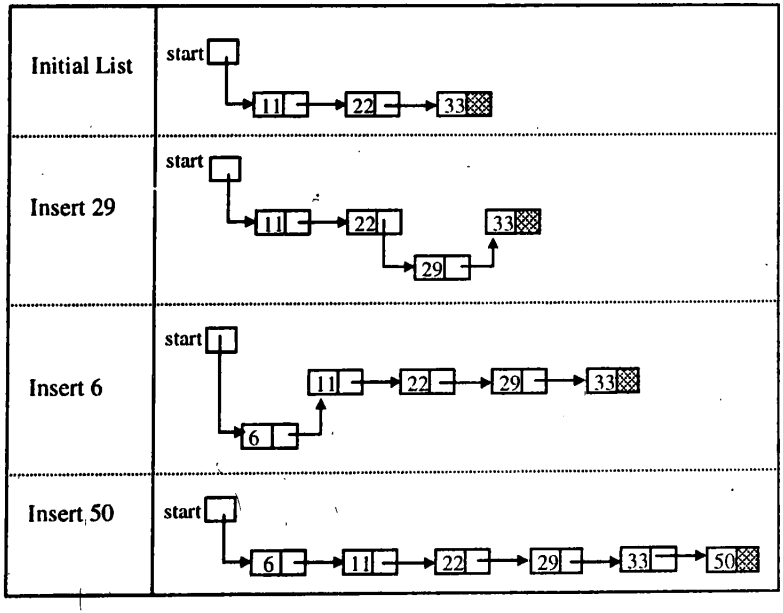


Figure 3.38 Insertion in a sorted linked list

When the element to be inserted is smaller than the first element, it will be added in the beginning. When the element to be inserted is greater than the last element, it will be added in the end. In other cases the element will be added in the list at its proper place.

We will make a function `insert_s()` that will insert an element in the list such that the ascending order is maintained. We have the same 4 cases as in other lists-

1. Insertion in the beginning.
2. Insertion in an empty list.
3. Insertion at the end.
4. Insertion in between.

Since we have taken a single linked list, insertion in beginning and insertion in an empty list can be handled in the same way.

```
if(start==NULL || data < start->info)
{
        tmp->link = start;
        start = tmp;
        return start;
}
```

For insertion in between, we will traverse the list and find a pointer to the node after which our new node should be inserted.

```
p = start;
while(p->link != NULL && p->link->info < data)
        p = p->link;
```

The new node has to be inserted after the node which is pointed by pointer p. The two lines of insertion are same as in single linked list.

```
tmp->link = p->link;
p->link = tmp;
```

If the insertion is to be done in the end, then also the above statements will work.

Other functions like `display()`, `count()` etc will remain same. The functions `search()` will be altered a little because here we can stop our search as soon as we find an element with value larger than the given element to be searched. The functions like `addatbeg()`, `addatend()`, `addafter()`, `addbefore()`, `addatpos()` don't make sense here because if we use these functions then the sorted order of the list might get disturbed. The function `insert_s()` decides where the element has to be inserted and inserts it in the proper place. The process of deletion is same as in single linked list.

```
/*P3.5 Program of sorted linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
struct node *insert_s(struct node *start,int data);
void search(struct node *start,int data);
void display(struct node *start);

main()
{
        int choice,data;
        struct node *start = NULL;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Display\n");
                printf("3.Search\n");
                printf("4.Exit\n");
                printf("Enter your choice : ");
```

```
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start = insert_s(start,data);
                        break;
                 case 2:
                        display(start);
                        break;
                 case 3:
                        printf("Enter the element to be searched : ");
                        scanf("%d",&data);
                        search(start,data);
                        break;
                 case 4:
                        exit(1);
                 default:
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
} /*end of main */
struct node *insert_s(struct node *start,int data)
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        /*list empty or new node to be added before first node*/
        if(start==NULL || data<start->info)
        {
                tmp->link = start;
                start = tmp;
                return start;
        }
        else
        {
                p = start;
                while(p->link!=NULL && p->link->info < data)
                        p = p->link;
                tmp->link = p->link;
                p->link = tmp;
        }
        return start;
}/*End of insert()*/
void search(struct node *start,int data)
{
        struct node *p;
        int pos;
        if(start==NULL || data < start->info)
        {
                printf("%d not found in list\n",data);
                return;
        }
        p = start;
        pos = 1;
        while(p!=NULL && p->info<=data)
        {
                if(p->info == data)
                {
                        printf("%d found at position %d\n",data,pos);
                        return;
                }
```

```
                p = p->link;
                pos++;
        }
        printf("%d not found in list\n",data);
}/*End of search()*/
void display(struct node *start)
{
        struct node *q;
        if(start == NULL)
        {
                printf("List is empty\n");
                return;
        }
        q = start;
        printf("List is :\n");
        while(q!=NULL)
        {
                printf("%d ",q->info);
                q = q->link;
        }
        printf("\n");
}/*End of display() */
```

3.6 Sorting a Linked List

If we have a linked list in unsorted order and we want to sort it, we can apply any sorting algorithm. We will use selection sort and bubble sort techniques (procedure given in chapter 8). In that chapter the elements to be sorted are stored in an array and here elements to be sorted are stored in a linked list. So here we can carry out the sorting in two ways-

(1) By exchanging the data
(2) By rearranging the links

Sorting by exchanging the data is similar to sorting carried out in arrays. If we have large records then this method is inefficient since the movement of records will take more time. In linked list, we can perform the sorting by one more method i.e. by rearranging the links. In this case there will be no movement of data, only the links will be changed.

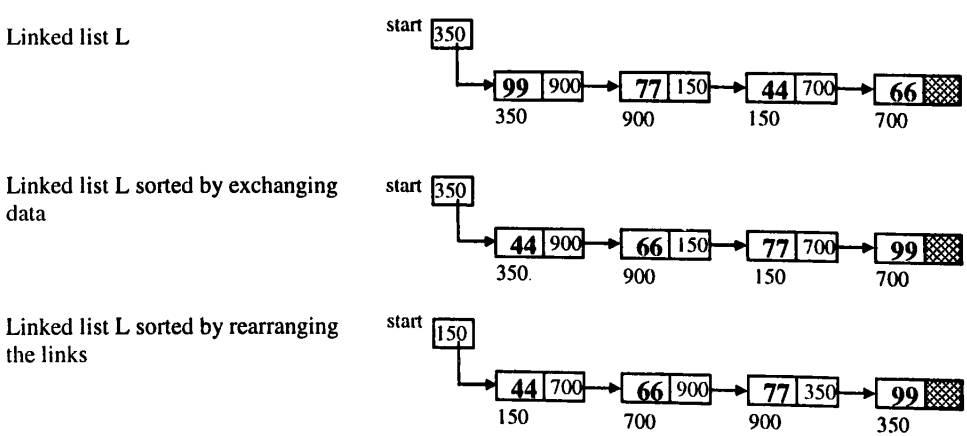


Figure 3.39

3.6.1 Selection Sort by exchanging data

The procedure of sorting a linked list through selection sort is shown in the following figure.

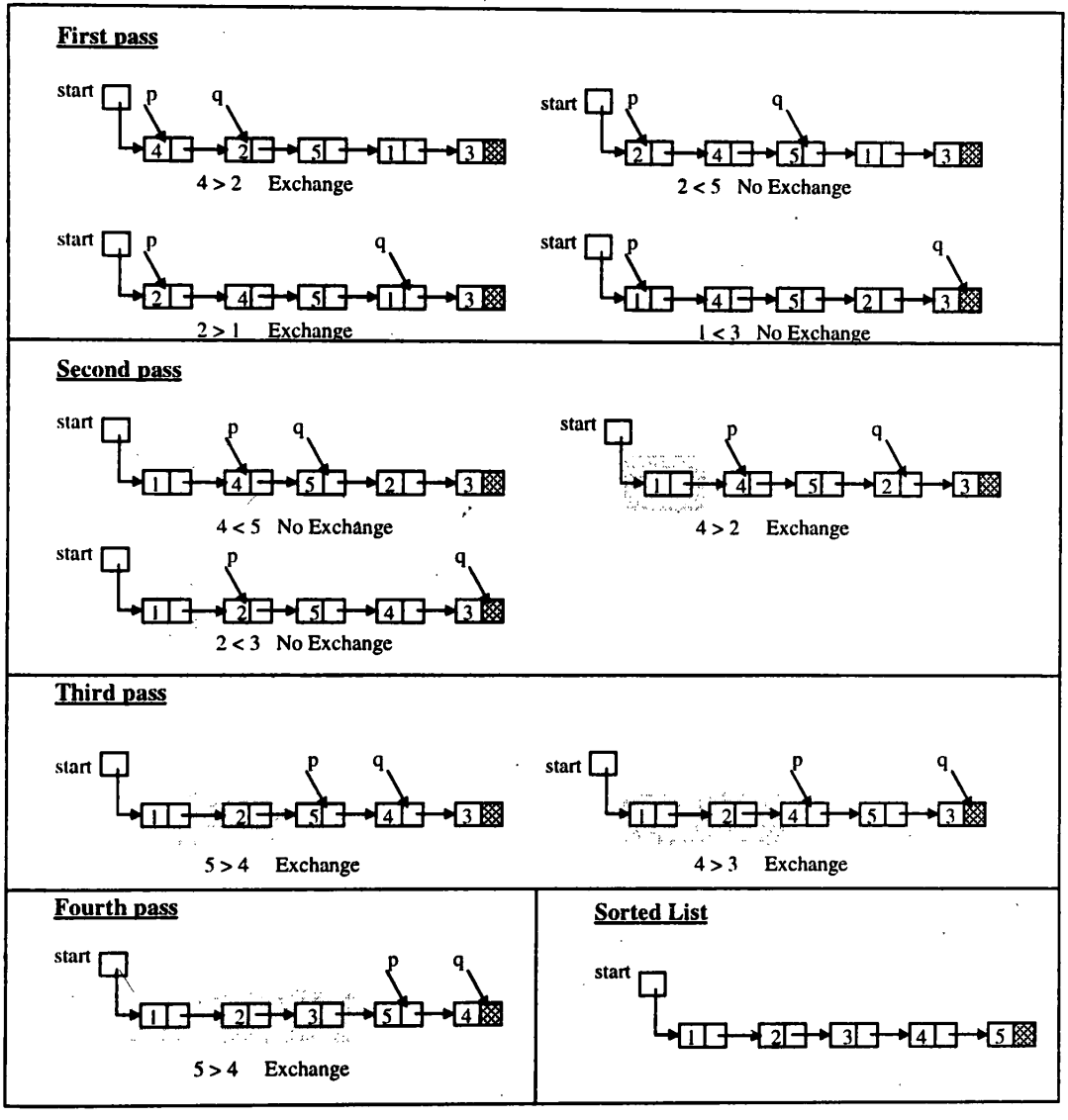


Figure 3.40 Sorting a Linked List using Selection sort

The following function will sort a single linked list through selection sort technique by exchanging data.

```
void selection(struct node *start)

    struct node *p,*q;
    int tmp;
    p = start;

    for(p=start; p->link!=NULL; p=p->link)
    {
            for(q=p->link; q!=NULL; q=q->link)
            {
```

```
                    if(p->info > q->info)
                    {
                            tmp = p->info;
                            p->info = q->info;
                            q->info = tmp;
                    }
            }
        }
}/*End of selection()*/
```

The terminating condition for outer loop is `(p->link!=NULL)`, so it will terminate when p points to the last node, i.e. it will work till p reaches second last node. The terminating condition for inner loop is `(q!=NULL)`, so it will terminate when q becomes NULL, i.e. it will work till q reaches last node. After each iteration of the outer loop, the smallest element from the unsorted elements will be placed at its proper place. In the figure 3.40, the shaded portion shows the elements that have been placed at their proper place.

3.6.2 Bubble Sort by exchanging data

The procedure of sorting a linked list through bubble sort is shown in the figure 3.41. After each pass, the largest element from the unsorted elements will be placed at its proper place. In the figure 3.41 the shaded portion shows the elements that have been placed at their proper place. In bubble sort, adjacent elements are compared so we will compare nodes pointed by pointers p and q where q is equal to `p->link`.

In each pass of the bubble sort, comparison starts from the beginning but the end changes each time. So in the inner loop, pointer p is always initialized to `start`. We have defined a pointer variable `end`, and the loop will terminate when link of p is equal to `end`. So the inner loop can be written as-

```
for(p=start; p->link!=end; p=p->link)
{
        q = p->link;
        if(p->info > q->info )
        {
                tmp = p->info;
                p->info = q->info;
                q->info = tmp;
        }
}
```

Now we have to see how the pointer variable `end` has to be initialized and changed. This will be done in the outer loop.

```
for(end=NULL; end!=start->link; end=q)
{
        for(p=start; p->link!=end; p=p->link)
        {
                q = p->link;
                if(p->info > q->info)
                {
                        tmp = p->info;
                        p->info = q->info;
                        q->info = tmp;
                }
        }
}
```

The pointer variable `end` is NULL in the first iteration of outer loop, so inner loop will terminate when p points to the last node i.e. the inner loop will work only till p reaches second last node. After first iteration, value of `end` is updated and is made equal to q. So now `end` points to the last node. This time the inner loop will terminate when p points to the second last node i.e. the inner loop will work only till p reaches third last node.

After each iteration of outer loop, the pointer `end` moves one node back towards the beginning. Initially `end` is NULL, after first iteration it points to the last node, after second iteration it points to the second last node and

so on. The terminating condition for outer loop is taken as `(end!=start->link)`, so the outer loop will terminate when `end` points to second node, i.e. the outer loop will work only till `end` reaches the third node.

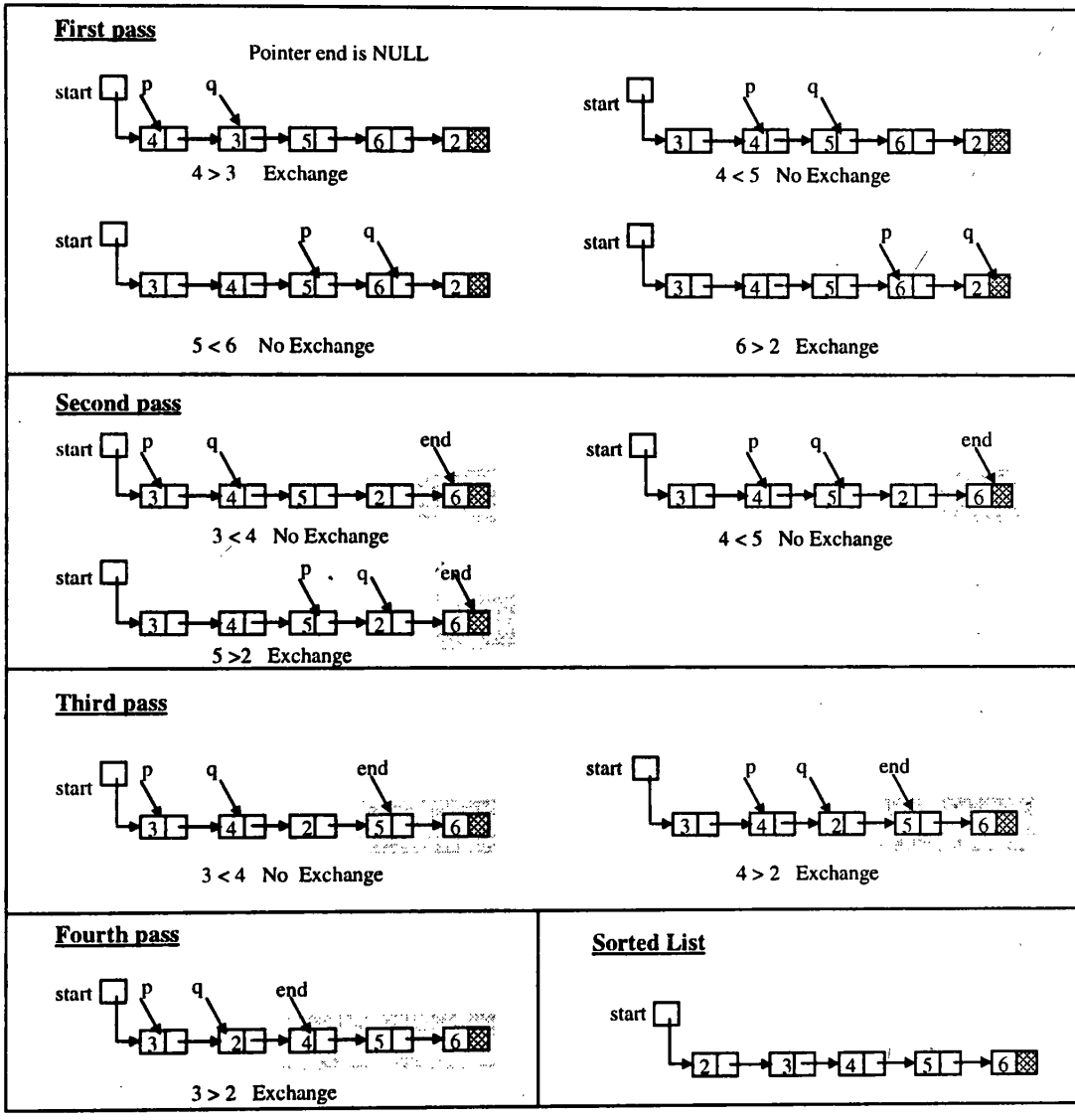


Figure 3.41 Sorting Linked List using Bubble Sort

The following function will sort a single linked list through bubble sort technique by exchanging data.

```
void bubble(struct node *start)
{
        struct node *end,*p,*q;
        int tmp;
        for(end=NULL; end!=start->link; end=q)
        {
                for(p=start; p->link!=end; p=p->link)
                {
                        q = p->link;
                        if(p->info > q->info)
                        {
                                tmp = p->info;
```

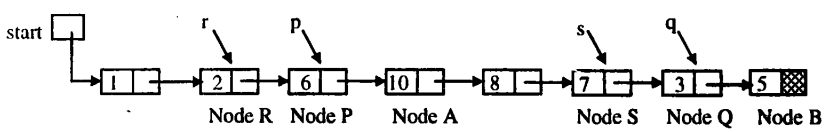
```
                        p->info = q->info;
                        q->info = tmp;
                    }
                }
            }
}/*End of bubble()*/
```

3.6.3 Selection Sort by rearranging links

The pointers p and q will move in the same manner as in selection sort by exchanging data. We will compare nodes pointed by pointers p and q. If the value in node pointed by p is more than the value in node pointed by q then we will have to change the links such that the positions of these nodes in the list are exchanged. For changing the positions we will need the address of predecessor nodes also. So we will take two more pointers r and s which will point to the predecessors of nodes pointed by p and q respectively. In this case the two loops can be written as-

```
for(r=p=start; p->link!=NULL; r=p,p=p->link)
{
        for(s=q=p->link; q!=NULL; s=q,q=q->link)
        {
                if(p->info > q->info)
                {
                        .....................
                }
        }
}
```

In both the loops, pointers p and q are initialized and changed in the same way as in selection sort by exchanging data. Now let us see what is to be done if p->info is greater then q->info.



The positions of nodes P and Q have to be exchanged, i.e. node P should be between nodes S and B and node Q should be between nodes R and A

(i) Node P should be before node B, so link of node P should point to node B
`p->link = q->link;`
(ii) Node Q should be before node A, so link of node Q should point to node A
`q->link = p->link;`
(iii) Node Q should be after node R, so link of node R should point to node Q
`r->link = q;`
(iv) Node P should be after node S, so link of node S should point to node P
`s->link = p;`

For writing the first two statements we will need a temporary pointer, since we are exchanging p->link and q->link.

```
tmp = p->link;
p->link = q->link;
q->link = tmp;
```

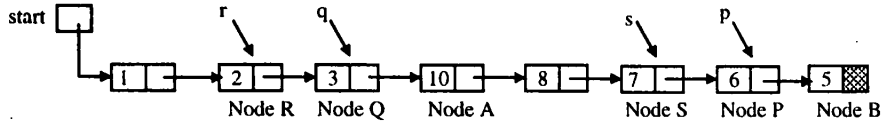
If p points to the first node, then r also points to the first node i.e. nodes R and P both are same, so in this case there is no need of writing the third statement (r->link = q;)
We need the third statement only if the pointer p is not equal to start. So it can be written as-

```
if(p!=start)
        r->link = q;
```

If start points to node P, then start needs to be updated and now it should point to node Q.

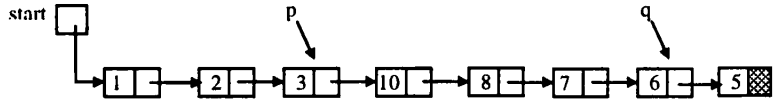
```
if(p==start)
        start = q;
```

After writing the above statements the linked list will look like this-



The positions of nodes P and Q have changed and this is what we want because the value in node P was more than value in node Q. Now we will bring the pointers p and q back to their positions to continue with our sorting process. For this we will exchange the pointers p and q with the help of a temporary pointer.

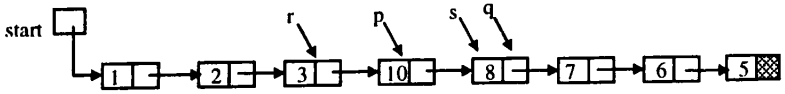
```
tmp = p;    p = q;   q = tmp;
```



Here is the function for selection sort by rearranging links.

```
struct node *selection_1(struct node *start)
{
        struct node *p,*q,*r,*s,*tmp;
        for(r=p=start; p->link!=NULL; r=p,p=p->link)
        {
                for(s=q=p->link; q!=NULL; s=q,q=q->link)
                {
                        if(p->info > q->info)
                        {
                                tmp = p->link;
                                p->link = q->link;
                                q->link = tmp;
                                if(p!=start)
                                r->link = q;
                                s->link = p;
                                if(p == start)
                                        start = q;
                                tmp = p;
                                p = q;
                                q = tmp;
                        }
                }
        }
        return start;
}/*End of selection_1()*/
```

In the previous figures, we have taken the case when p and q point to non adjacent nodes, and we have written our code according to this case only. Now let us see whether this code will work when p and q point to adjacent nodes. The pointers p and q will point to adjacent nodes only in the first iteration of inner loop. In that case s and q point to same node. The following figure shows this situation-



You can see that the code we have written will work in this case also. So there is no need to consider a separate case when nodes p and q are adjacent.

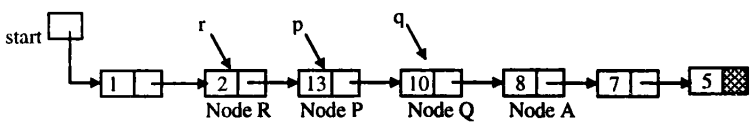
When we sort a linked list by exchanging data, links are not disturbed so `start` remains same. But when sorting is done by rearranging links, the value of `start` might change so it is necessary to return the value of `start` at the end of the function.

3.6.4 Bubble sort by rearranging links

In bubble sort, since `p` and `q` are always adjacent, there is no need of taking predecessor of node pointed by `q`. Both the loops are written in the same way as in bubble sorting by exchanging data. In the inner loop we have taken a pointer `r` that will point to the predecessor of node pointed by `p`.

```
for(end=NULL; end!=start->link; end=q)
{
        for(r=p=start; p->link!=end; r=p,p=p->link)
        {
                q = p->link;
                if(p->info > q->info )
                {
                        .....
                }
        }
}
```

Now let us see what is to be done if `p->info` is greater than `q->info`.



Node P should be before node A, so link of node P should point to node A

```
p->link = q->link;
```

Node Q should be before node P, so link of node Q should point to node P

```
q->link = p;
```

Node Q should be after node R, so link of node R should point to node Q

```
r->link = q;
```

Here is the function for bubble sort by rearranging links.

```
struct node *bubble_l(struct node *start)
{
        struct node *end,*r,*p,*q,*tmp;
        for(end=NULL; end!=start->link; end=q)
        {
                for(r=p=start; p->link!=end; r=p,p=p->link)
                {
                        q = p->link;
                        if(p->info > q->info )
                        {
                                p->link = q->link;
                                q->link = p;
                                if(p!=start)
                                  r->link = q;
                                else
                                    start = q;
                                tmp = p;
                                p = q;
                                q = tmp;
                        }
                }
        }
        return start;
}/*End of bubble_l()*/
```

3.7 Merging

If there are two sorted linked lists, then the process of combining these sorted lists into another list of sorted order is called merging. The following figure shows two sorted lists and a third list obtained by merging them.

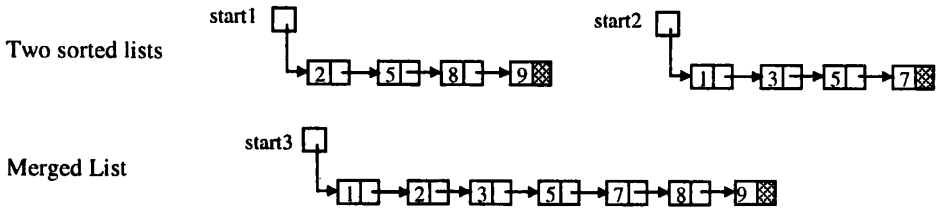


Figure 3.42

If there is any element that is common to both the lists, it will be inserted only once in the third list. For merging, both the lists are scanned from left to right. We will take one element from each list, compare them and then take the smaller one in third list. This process will continue until the elements of one list are finished. Then we will take the remaining elements of unfinished list in third list. The whole process for merging is shown in the figure 3.43. We've taken two pointers `p1` and `p2` that will point to the nodes that are being compared. There can be three cases while comparing `p1->info` and `p2->info`

1. If `(p1->info) < (p2->info)`

The new node that is added to the resultant list has info equal to `p1->info`. After this we will make `p1` point to the next node of first list.

2. If `(p2->info) < (p1->info)`

The new node that is added to the resultant list has info equal to `p2->info`. After this we will make `p2` point to the next node of second list.

3. If `(p1->info) == (p2->info)`

The new node that is added to the resultant list has info equal to `p1->info`(or `p2->info`). After this we will make `p1` and `p2` point to the next nodes of first list and second list respectively. The procedure of merging is shown in figure 3.43.

```
while(p1!=NULL && p2!=NULL)
{
        if(p1->info < p2->info)
        {
                start3 = insert(start3,p1->info);
                p1 = p1->link;
        }
        else if(p2->info < p1->info)
        {
                start3 = insert(start3,p2->info);
                p2 = p2->link;
        }
        else if(p1->info == p2->info)
        {
                start3 = insert(start3,p1->info);
                p1 = p1->link;
                p2 = p2->link;
        }
}
```

The above loop will terminate when any of the list will finish. Now we have to add the remaining nodes of the unfinished list to the resultant list. If second list has finished, then we will insert all the nodes of first list in the resultant list as-

```
while(p1!=NULL)
{
        start3 = insert(start3,p1->info);
```

```
        p1 = p1->link;
}
```

If first list has finished, then we will insert all the nodes of second list in the resultant list as-

```
while(p2!=NULL)
{
        start3 = insert(start3,p2->info);
        p2 = p2->link;
}
```

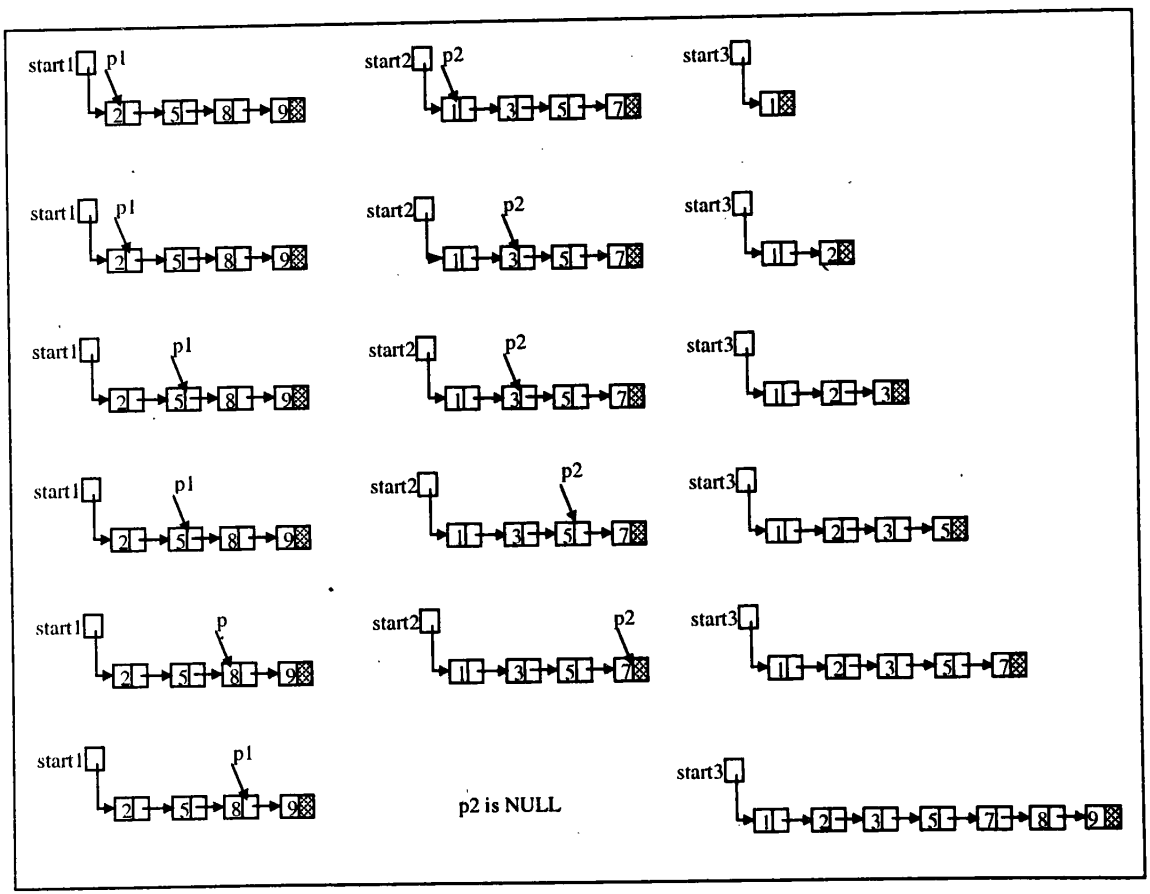


Figure 3.43 Merging two sorted linked lists

The program for merging is given next.

```
/*P3.7 Program of merging two sorted single linked lists*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
struct node *create(struct node *start);
struct node *insert_s(struct node *start,int data);
struct node *insert(struct node *start,int data);
void display(struct node *start);
void merge(struct node *p1,struct node *p2);
```

```
main()
{
        struct node *start1 = NULL,*start2 = NULL;
        start1 = create(start1);
        start2 = create(start2);
        printf("List1 : "); display(start1);
        printf("List2 : "); display(start2);
        merge(start1, start2);
}/*End of main()*/

void merge(struct node *p1,struct node *p2)
{
        struct node *start3;
        start3 = NULL;
        while(p1!=NULL && p2!=NULL)
        {
                if(p1->info < p2->info)
                {
                        start3 = insert(start3,p1->info);
                        p1 = p1->link;
                }
                else if(p2->info < p1->info)
                {
                        start3 = insert(start3,p2->info);
                        p2 = p2->link;
                }
                else if(p1->info==p2->info)
                {
                        start3 = insert(start3,p1->info);
                        p1 = p1->link;
                        p2 = p2->link;
                }
        }
        /*If second list has finished and elements left in first list*/
        while(p1!=NULL)
        {
                start3 = insert(start3,p1->info);
                p1 = p1->link;
        }
        /*If first list has finished and elements left in second list*/
        while(p2!=NULL)
        {
                start3 = insert(start3,p2->info);
                p2 = p2->link;
        }
        printf("Merged list is : ");
        display(start3);
}

struct node *create(struct node *start)
{
        int i,n,data;

        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        start = NULL;
        for(i=1;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                start = insert_s(start, data);
        }
        return start;
}/*End of create_slist()*/

struct node *insert_s(struct node *start,int data)
```

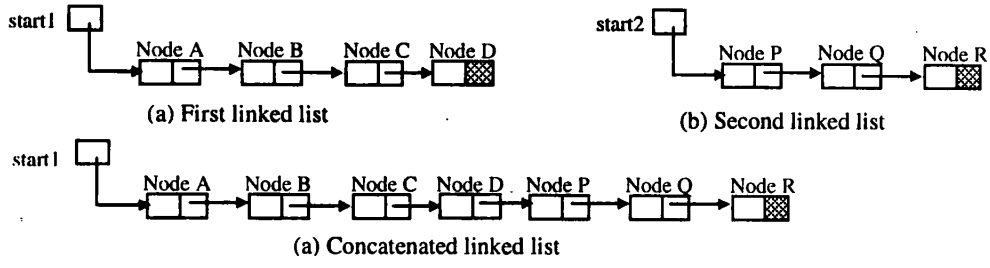
```
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        /*list empty or data to be added in beginning */
        if(start == NULL || data<start->info)
        {
                tmp->link = start;
                start = tmp;
                return start;
        }
        else
        {
                p = start;
                while(p->link!=NULL && p->link->info < data)
                        p = p->link;
                tmp->link = p->link;
                p->link = tmp;
        }
        return start;
}/*End of insert_s()*/

struct node *insert(struct node *start,int data)
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        if(start == NULL)  /*If list is empty*/
        {
                tmp->link = start;
                start = tmp;
                return start;
        }
        else    /*Insert at the end of the list*/
        {
                p = start;
                while(p->link!=NULL)
                        p = p->link;
                tmp->link = p->link;
                p->link = tmp;
        }
        return start;
}/*End of insert()*/

void display(struct node *start)
{
        struct node *p;
        if(start == NULL)
        {
                printf("List is empty\n");
                return;
        }
        p = start;
        while(p!=NULL)
        {
                printf("%d ",p->info);
                p = p->link;
        }
        printf("\n");
}/*End of display()*/
```

The function `insert_s()` is the same that we have made in sorted linked list and it inserts nodes in ascending order. We have used this function to create the two sorted lists that are to be merged. The function

`insert()` is a simple function that inserts nodes in a linked list at the end. We have used this function to insert nodes in the third list.

3.8 Concatenation

Suppose we have two single linked lists and we want to append one at the end of another. For this the link of last node of first list should point to the first node of the second list. Let us take two single linked lists and concatenate them.



(a) First linked list

(b) Second linked list

(a) Concatenated linked list

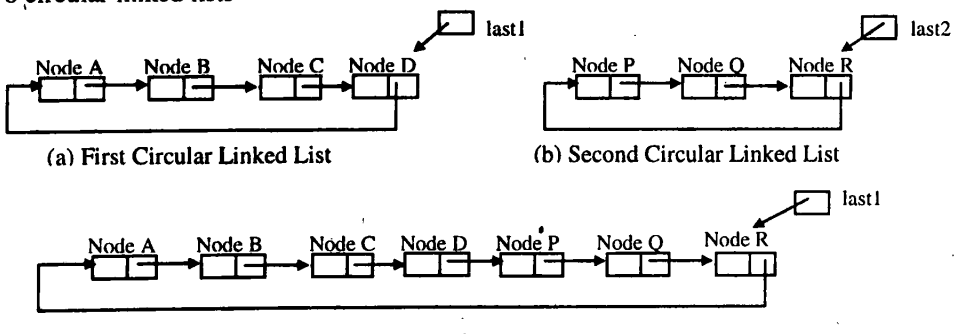
Figure 3.44

For concatenation, link of node D should point to node P. To get the address of node D, we have to traverse the first list till the end. Suppose `ptr` points to node D, then `ptr->link` should be made equal to `start2`.

```
ptr->link = start2;
```

```
struct node *concat(struct node *start1, struct node *start2)
{
        struct node *ptr;
        if(start1 == NULL)
        {
                start1 = start2;
                return start1;
        }
        if(start2 == NULL)
                return start1;
        ptr = start1;
        while(ptr->link != NULL)
                ptr = ptr->link;
        ptr->link = start2;
        return start1;
}
```

If we have to concatenate two circular linked lists, then there is no need to traverse any of the lists. Let us take two circular linked lists-



(a) First Circular Linked List

(b) Second Circular Linked List

(c) Concatenated list

Figure 3.45

Link of node D should point to node P

```
last1->link = last2->link;
```

We will lose the address of node A, so before writing this statement we should save `last1->link`.

```
ptr = last1->link;
```

Link of node R should point to node A

```
last2->link = ptr;
```

The pointer `last1` should point to node R

```
last1 = last2;
```

```
struct node *concat(struct node *last1, struct node *last2)
{
        struct node *ptr;
        if(last1 == NULL)
        {
                last1 = last2;
                return last1;
        }
        if(last2 == NULL)
                return last1;
        ptr = last1->link;
        last1->link = last2->link;
        last2->link = ptr;
        last1 = last2;
        return last1;
}
```

3.9 Polynomial arithmetic with linked list

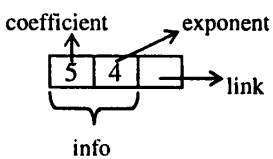
A useful application of linked list is the representation of polynomial expressions. Let us take a polynomial expression with single variable-

$5x^4 + x^3 - 6x + 2$

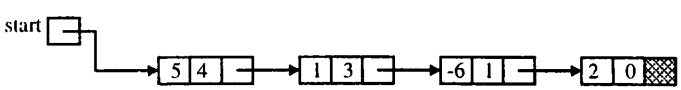
In each term we have a coefficient and an exponent. For example in the term $5x^4$, coefficient is 5 and exponent is 4. The whole polynomial can be represented through linked list where each node will represent a term of the expression. The structure for each node will be-

```
struct node{
                float coefficient;
                int exponent;
                struct node *link;
            }
```

Here the info part of the node contains coefficient and exponent and the link part is same as before and will be used to point to the next node of the list. The node representing the term $5x^4$ can be represented as-



The polynomial $(5x^4 + x^3 - 6x + 2)$ can be represented through linked list as-



Here 2 is considered as $2x^0$ because $x^0 = 1$.

The arithmetic operations are easier if the terms are arranged in descending order of their exponents. For example it would be better if the polynomial expression $(5x + 6x^3 + x^2 - 9 + 2x^6)$ is stored as $(2x^6 + 6x^3 + x^2 + 5x - 9)$. So for representing the polynomial expression, we will use sorted linked list which would be in

descending order based on the exponent. An empty list will represent zero polynomial. The following program shows creation of polynomial linked lists and their addition and multiplication.

```
/*P3.10 Program of polynomial addition and multiplication using linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        float coef;
        int expo;
        struct node *link;
};
struct node *create(struct node *);
struct node *insert_s(struct node *,float,int);
struct node *insert(struct node *,float,int);
void display(struct node *ptr);
void poly_add(struct node *,struct node *);
void poly_mult(struct node *,struct node *);

main()
{
        struct node *start1 = NULL,*start2 = NULL;
        printf("Enter polynomial 1 :\n");   start1 = create(start1);
        printf("Enter polynomial 2 :\n");   start2 = create(start2);
        printf("Polynomial 1 is :  ");  display(start1);
        printf("Polynomial 2 is :  ");  display(start2);
        poly_add(start1, start2);
        poly_mult(start1, start2);
}/*End of main()*/

struct node *create(struct node *start)
{
        int i,n,ex;
        float co;
        printf("Enter the number of terms : ");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("Enter coeficient for term %d : ",i);
                scanf("%f",&co);
                printf("Enter exponent for term %d : ",i);
                scanf("%d",&ex);
                start = insert_s(start,co,ex);
        }
        return start;
}/*End of create()*/

struct node *insert_s(struct node *start,float co,int ex)
{
        struct node *ptr,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->coef = co;
        tmp->expo = ex;
        /*list empty or exp greater than first one*/
        if(start == NULL || ex > start->expo)
        {
                tmp->link = start;
                start = tmp;
        }
        else
        {
                ptr = start;
                while(ptr->link!=NULL && ptr->link->expo >= ex)
                        ptr = ptr->link;
```

```
                tmp->link = ptr->link;
                ptr->link = tmp;
        }
        return start;
}/*End of insert()*/

struct node *insert(struct node *start,float co,int ex)
{
        struct node *ptr,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->coef = co;
        tmp->expo = ex;
        if(start == NULL)       /*If list is empty*/
        {
                tmp->link = start;
                start = tmp;
        }
        else    /*Insert at the end of the list*/
        {
                ptr = start;
                while(ptr->link!=NULL)
                        ptr = ptr->link;
                tmp->link = ptr->link;
                ptr->link = tmp;
        }
        return start;
}/*End of insert()*/

void display(struct node *ptr)
{
        if(ptr == NULL)
        {
                printf("Zero polynomial\n");
                return;
        }
        while(ptr!=NULL)
        {
                printf("(%.1fx^%d)", ptr->coef,ptr->expo);
                ptr = ptr->link;
                if(ptr!=NULL)
                        printf(" + ");
                else
                        printf("\n");
        }
}/*End of display()*/

void poly_add(struct node *p1,struct node *p2)
{
        struct node *start3;
        start3 = NULL;
        while(p1!=NULL && p2!=NULL)
        {
                if(p1->expo > p2->expo)
                {
                        start3 = insert(start3,p1->coef,p1->expo);
                        p1 = p1->link;
                }
                else if(p2->expo > p1->expo)
                {
                        start3 = insert(start3,p2->coef,p2->expo);
                        p2 = p2->link;
                }
                else if(p1->expo == p2->expo)
                {
                        start3 = insert(start3,p1->coef+p2->coef,p1->expo);
```

```
                p1 = p1->link;
                p2 = p2->link;
            }
        }
        /*if poly2 has finished and elements left in poly1*/
        while(p1!=NULL)
        {
            start3 = insert(start3,p1->coef,p1->expo);
            p1 = p1->link;
        }
        /*if poly1 has finished and elements left in poly2*/
        while(p2!=NULL)
        {
            start3 = insert(start3,p2->coef,p2->expo);
            p2 = p2->link;
        }
        printf("Added polynomial is : ");
        display(start3);
}/*End of poly_add() */

void poly_mult(struct node *p1, struct node *p2)
{
        struct node *start3;
        struct node *p2_beg = p2;

        start3 = NULL;
        if(p1 == NULL || p2 == NULL)
        {
            printf("Multiplied polynomial is zero polynomial\n");
            return;
        }
        while(p1!=NULL)
        {
           p2 = p2_beg;
           while(p2!=NULL)
           {
             start3 = insert_s(start3,p1->coef*p2->coef,p1->expo+p2->expo);
             p2 = p2->link;
           }
           p1 = p1->link;
        }
        printf("Multiplied polynomial is : ");
        display(start3);
}/*End of poly_mult()*/
```

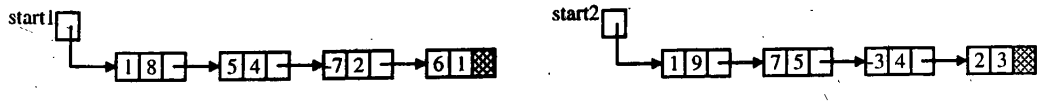
3.9.1 Creation of polynomial linked list

The function `create()` is very simple and calls another function `insert_s()` that inserts a node in polynomial linked list. The function `insert_s()` is similar to that of sorted linked lists. The only difference is that here our list is in descending order based on the exponent.

3.9.2 Addition of 2 polynomials

The procedure for addition of 2 polynomials represented by linked lists is somewhat similar to that of merging. Let us take two polynomial expression lists and make a third list by adding them.

Poly1- $x^8 + 5x^4 - 7x^2 + 6x$ Poly2- $x^9 + 7x^5 - 3x^4 + 2x^3$



The pointers p1 and p2 will point to the current nodes in the polynomials which will be added. The process of addition is shown in the figure 3.46. Both polynomials are traversed until one polynomial finishes. We can have three cases-

1. If (p1->expo) > (p2->expo)
The new node that is added to the resultant list has coefficient equal to p1->coef and exponent equal to p1->expo. After this we will make p1 point to the next node of polynomial1.

2. If (p2->expo) > (p1->expo)
The new node that is added to the resultant list has coefficient equal to p2->coef and exponent equal to p2->expo. After this we will make p2 point to the next node of polynomial2.

3. If (p1->expo) == (p2->expo)
The new node that is added to the resultant list has coefficient equal to (p1->coef + p2->coef) and exponent equal to p1->expo(or p2->expo). After this we will make p1 and p2 point to the next nodes of polynomial1 and polynomial 2 respectively. The procedure of polynomial addition is shown in figure 3.46.

```
while(p1!=NULL && p2!=NULL)
{
        if(p1->expo > p2->expo)
        {
                p3_start = insert(p3_start,p1->coef,p1->expo);
                p1 = p1->link;
        }
        else if(p2->expo > p1->expo)
        {
                p3_start = insert(p3_start,p2->coef,p2->expo);
                p2 = p2->link;
        }
        else if(p1->expo == p2->expo)
        {
                p3_start = insert (p3_start,p1->coef+p2->coef,p1->expo);
                p1 = p1->link;
                p2 = p2->link;
        }
}
```

The above loop will terminate when any of the polynomial will finish. Now we have to add the remaining nodes of the unfinished polynomial to the resultant list. If polynomial 2 has finished, then we will put all the terms of polynomial 1 in the resultant list as-

```
while(p1!=NULL)
{
        p3_start = insert(p3_start,p1->coef,p1->expo);
        p1 = p1->link;
}
```

If polynomial 1 has finished, then we will put all the terms of polynomial 2 in the resultant list as-

```
while(p2!=NULL)
{
        p3_start = insert(p3_start,p2->coef,p2->expo);
        p2 = p2->link;
}
```

We can see the advantage of storing the terms in descending order of their exponents. If it was not so then we would have to scan both the lists many times.

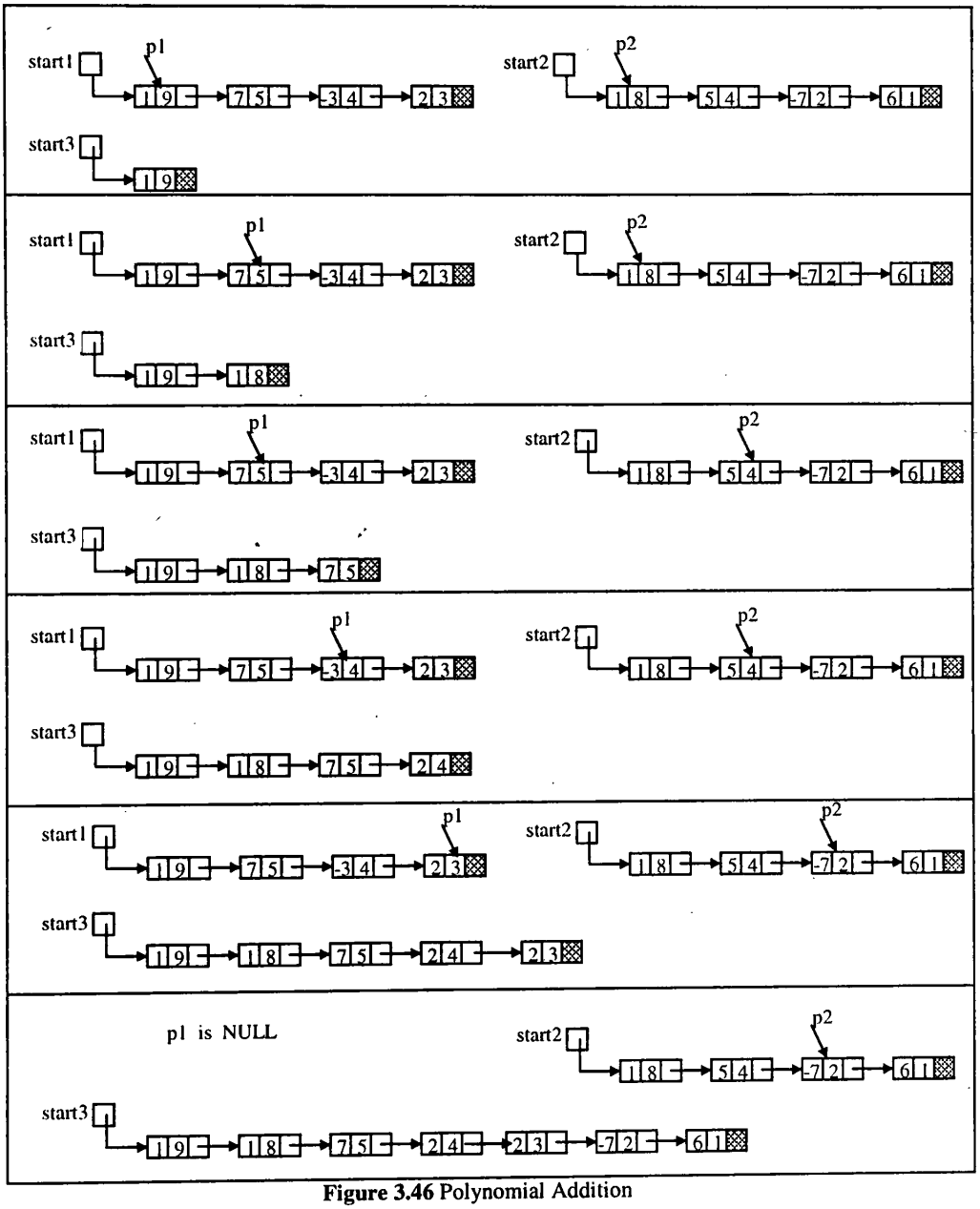


Figure 3.46 Polynomial Addition

3.9.3 Multiplication of 2 polynomials

Suppose we have to multiply the two polynomials given below-

$4x^3 + 5x^2 - 3x$

$2x^5 + 6x^4 + x^2 + 8$

For this we have to multiply each term of the first polynomial with each term of second polynomial. When we multiply two terms, their coefficients are multiplied and exponents are added. The product of the above two polynomials is-

$[(4*2)x^{3+5} + (4*6)x^{3+4} + (4*1)x^{3+2} + (4*8)x^{3+0}] + [(5*2)x^{2+5} (5*6)x^{2+4} +(5*1)x^{2+2} +(5*8)x^{2+0}] +$
$[(-3*2)x^{1+5} + (-3*6)x^{1+4} + (-3*1)x^{1+2} + (-3*8)x^{1+0}]$

$8x^8 + 24x^7 + 4x^5 + 32x^3 + 10x^7 + 30x^6 + 5x^4 + 40x^2 - 6x^6 - 18x^5 - 3x^3 - 24x^1$

It is obvious that we will have to use two nested loops, the outer loop will walk through the first polynomial and the inner loop will walk through the second polynomial.

In the function `poly_mult()`, we have used the function `insert_s()` to insert elements in the third list. If we don't do so, then the elements in the multiplied list will not be in descending order based on the exponent.

We have seen how to represent polynomial expressions with single variable through linked list. We can extend this concept to represent expressions with multiple variables. For example, the structure of a node that can be used to represent expressions with three variables is-

```
struct node
{
        float coef;
        int expx;
        int expy;
        int expz;
        struct node *link;
}
```

Another useful application of linked list is in radix sort which is explained in the chapter on Sorting.

3.10 Comparison of Array lists and Linked lists

Now after having studied about linked lists, let us compare the two implementations of lists i.e. the array implementation and the linked implementation.

3.10.1 Advantages of linked lists

(1) We know that the size of array is specified at the time of writing the program. This means that the memory space is allocated at compile time, and we can't increase or decrease it during runtime according to our needs. If the amount of data is less than the size of array then space is wasted, while if data is more than size of array then overflow occurs even if there is enough space available in memory.

Linked lists overcome this problem by using dynamically allocated memory. The size of linked list is not fixed and it can be increased or decreased during runtime. Since memory is allocated at runtime, there is no wastage of memory and we can keep on inserting the elements till memory is available. Whenever we need a new node we dynamically allocate it i.e. we get it from the free storage space. When we don't need the node, we can return the memory occupied by it to the free storage space so that it can be used by other programs. We have done these two operations by using `malloc()` and `free()`.

(2) Insertion and deletion inside arrays is not efficient since it requires shifting of elements. For example if we want to insert an element at the 0^{th} position then we have to shift all the elements of the array to the right, and if we want to delete the element present at the 0^{th} position then we have to shift all the elements to the left. These are the worst cases of insertion and deletion and the efficiency is O(n). If the array consists of big records then this shifting is even more time consuming.

In linked lists, insertion and deletion requires only change in pointers. There is no physical movement of data, only the links are altered.

(3) We know that in arrays all the elements are always stored in contiguous locations. Sometimes arrays can't be used because the amount of memory needed by them is not available in contiguous locations i.e. the total

memory required by the array is available but it is dispersed. So in this case we can't create an array in spite of available memory.
In linked list, elements are not stored in contiguous locations. So in the above case, there will be no problem in creation of linked list.

3.10.2 Disadvantages of linked lists

(1) In arrays we can access the n^{th} element directly, but in linked lists we have to pass through the first n-1 elements to reach the n^{th} element. So when it comes to random access, array lists are definitely better than linked lists.
(2) In linked lists the pointer fields take extra space that could have been used for storing some more data.
(3) Writing of programs for linked list is more difficult than that for arrays.

Exercise

In all the problems assume that we have an integer in the info part of nodes.

1. Write a function to count the number of occurrences of an element in a single linked list.
2. Write a function to find the smallest and largest element of a single linked list.
3. Write a function to check if two linked lists are identical. Two lists are identical if they have same number of elements and the corresponding elements in both lists are same
4. Write a function to create a copy of a single linked list.
5. Given a linked list L, write a function to create a single linked list that is reverse of the list L. For example if the list L is `1->2->3->4->5` then the new list should be `5->4->3->2->1`. The list L should remain unchanged.
6. Write a program to swap adjacent elements of a single linked list
(i) by exchanging info part
(ii) by rearranging links.
For example if a linked list is `1->2->3->4->5->6->7->8`, then after swapping adjacent elements it should become `2->1->4->3->6->5->8->7`.
7. Write a program to swap adjacent elements of a double linked list by rearranging links.
8. Write a program to swap the first and last elements of a single linked list
(i) by exchanging info part.
(ii) by rearranging links.
9. Write a function to move the largest element to the end of a single linked list.
10. Write a function to move the smallest element to the beginning of a single linked list.
11. Write a function for deleting all the nodes from a single linked list which have a value N.
12. Given a single linked list L1 which is sorted in ascending order, and another single linked list L2 which is not sorted, write a function to print the elements of second list according to the first list. For example if the first list is `1->2->5->7->8`, then the function should print the 1^{st}, 2^{nd}, 5^{th}, 7^{th}, 8^{th} elements of second list.
13. Write a program to remove first node of the list and insert it at the end, without changing info part of any node.
14. Write a program to remove the last node of the list and insert it in the beginning, without changing info part of any node.
15. Write a program to move a node n positions forward in a single linked list.
16. Write a function to delete a node from a single linked list. The only information we have is a pointer to the node that has to be deleted.
17. Write functions to insert a node just before and just after a node pointed to by a pointer p, without using the pointer start.

18. What is wrong in the following code that attempts to free all the nodes of a single linked list.

```
p=start;
while(p!=NULL)
{
        free(p);
        p=p->link;
}
```

Write a function `Destroy()` that frees all the nodes of a single linked list.

19. Write a function to remove duplicates from a sorted single linked list.

20. Write a function to remove duplicates from an unsorted single linked list.

21. Write a function to create a linked list that is intersection of two single linked lists, i.e. it contains only the elements which are common to both the lists.

22. Write a function to create a linked list that is union of two single linked lists, i.e. it contains all elements of both lists and if an element is repeated in both lists, then it is included only once.

23. Given a list L1, delete all the nodes having negative numbers in info part and insert them into list L2 and all the nodes having positive numbers into list L3. No new nodes should be allocated.

24. Given a linked list L1, create two linked lists one having the even numbers of L1 and the other having the odd numbers of L1. Don't change the list L1.

25. Write a function to delete alternate nodes(even numbered nodes) from a single linked list. For example if the list is `1->2->3->4->5->6->7` then the resulting list should be `1->3->5->7`.

26. Write a function to get the n^{th} node from the end of a single linked list, without counting the elements or reversing the list.

27. Write a function to find out whether a single linked is NULL terminated or contains a cycle/loop. If the list contains a cycle, find the length of the cycle and the length of the whole list. Find the node that causes the cycle i.e. the node at which the cycle starts. This node is pointed by two different nodes of the list. Remove the cycle from the list and make it NULL terminated.

28. Write a function to find out the middle node of a single linked list without counting all the elements of the list.

29. Write a function to split a single linked list into two halves.

30. Write a function to split a single linked list into two lists at a node containing the given information.

31. Write a function to split a single linked list into two lists such that the alternate nodes(even numbered nodes) go to a new list.

32. Write a function to combine the alternate nodes of two null terminated single linked lists. For example if the first list is `1->2->3->4` and the second list is `5->7->8->9` then after combining them the first list should be `1->5->2->7->3->8->4->9` and second list should be empty. If both lists are not of the same length, then the remaining nodes of the longer list are taken in the combined list. For example if the first list is `1->2->3->4` and the second list is `5->7` then the combined list should be `1->5->2->7->3->4`.

33. Suppose there are two null terminated single linked lists which merge at a given point and share all the nodes after that merge point (Y shaped lists). Write a function to find the merge point (intersection point).

34. Create a double linked list in which info part of each node contains a digit of a given number. The digits should be stored in reverse order, i.e. the least significant digit should be stored in the first node and the most significant digit in the last node. If the number is 5468132 then the linked list should be `2->3->1->8->6->4->5`. Write a function to add two numbers represented by linked lists.

35. Modify the program in the previous problem so that now in each node of the list we can store 4 digits of the given number. For example if the number is 23156782913287 then the linked list would be `3287->8291->1567->23`.

36. Write a function to find whether a linked list is palindrome or not.

37. Construct a linked list in which each node has the following information about a student - rollno, name, marks in 3 subjects. Enter records of different students in list. Traverse this list and calculate the total marks, percentage of each student. Count the number of students who scored passing marks(above 40 percent).

38. Modify the previous program so that now the names are inserted in alphabetical order in the list. Make this program menu driven with the following menus-

(i) Create list (ii) Insert (iii) Delete (iv) Modify (v) Display record (vi) Display result
Delete menu should have the facility of entering name of a student and the record of that student should be deleted. Display record menu should ask for the roll no of a student and display all information. Display result should display the number of students who have passed. Modify menu has the facility of modifying a record given the roll number.

Stacks and Queues

In linked list and arrays, insertions and deletions can be performed at any place of the list. There can be situations when there is a need of a data structure in which operations are allowed only on the ends of the list and not in the middle. Stack and Queue are data structures which fulfill these requirements. Stack is a linear list in which insertions and deletions are allowed only at one end while queue is a linear list in which insertion is performed on one end and deletion is performed on the other end.

4.1 Stack

Stack is a linear list in which insertions and deletions are allowed only at one end, called top of the stack. We can see examples of stack in our daily life like stack of trays in a cafeteria, stack of books or stack of tennis balls. In all these cases we can see that any object can be removed or added only at the top.

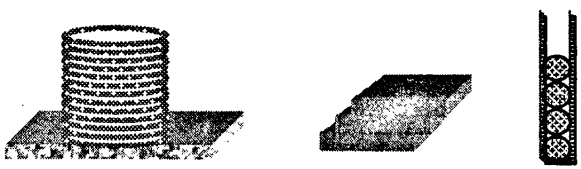


Figure 4.1 Examples of Stack

The insertion and deletion operations are given special names in the case of stack. The **push** operation inserts an element in the stack and **pop** operation deletes an element from the stack. The figure 4.2 shows these operations with the help of an example-

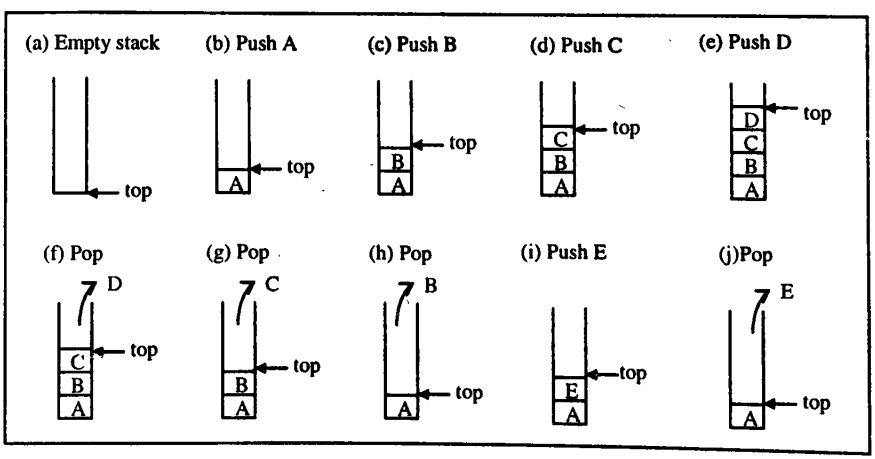


Figure 4.2

We can see that the element which is pushed last is popped first from the stack. In the example of figure 4.2, D is pushed at last but it was the first one to be popped. The behaviour of stack is like last in first out, so it is also called LIFO(Last In First Out) data structure.

Before pushing any element we must check whether there is space in the stack or not. If there is not enough space then stack is said to be in **overflow** state and the new element can't be pushed. Similarly before pop operation if stack is empty and pop operation is attempted, then stack is said to be in **underflow** state.

Since stack is a linear list, it can be implemented using arrays or linked lists. In the next two sections we will study these two implementations of stack.

4.1.1 Array Implementation of Stack

We will take a one-dimensional array `stack_arr[]` to hold the elements of the stack. In an array, elements can be added or deleted at any place but since we are implementing stack, we have to permit insertions and deletions at the top of the stack only. So we take a variable `top`, which keeps the position(index) of the topmost element in array.

Initially when the stack is empty, value of `top` is initialized to -1. For push operation, first the value of `top` is increased by 1 and then the new element is pushed at the position of `top`. For pop operation, first the element at the position of `top` is popped and then `top` is decreased by 1.

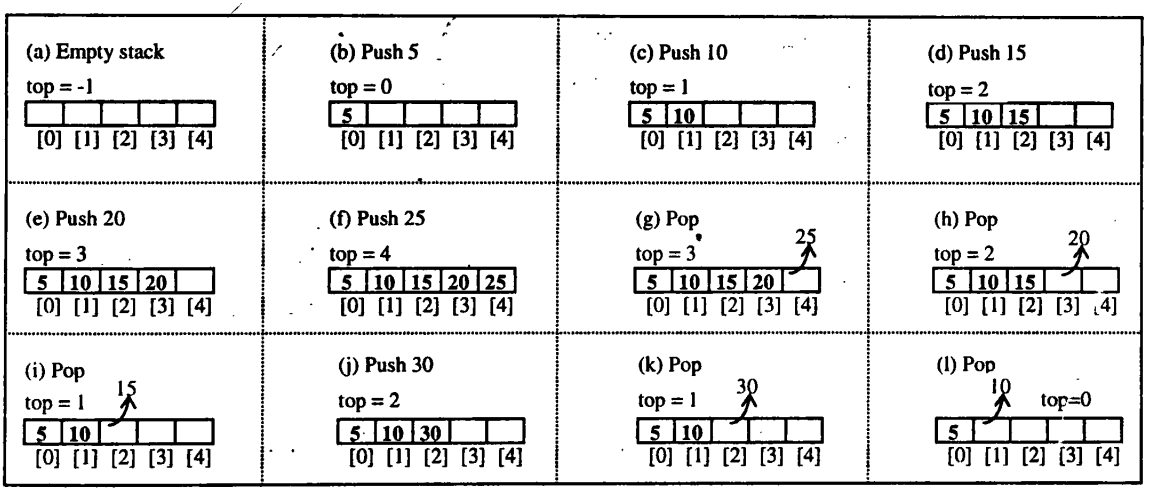


Figure 4.3

While writing functions for push and pop operations we have to check for overflow and underflow. If MAX is the size of array, then stack will become full when `top` becomes equal to MAX-1. In the figure, stack is full in step (f), and the value of `top` is 4. So before pushing we have to check for this overflow condition. The stack becomes empty when the value of `top` is -1, so we have to check this underflow condition before popping any element from the stack.

```
/*P4.1 Program of stack using array*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 10
int stack_arr[MAX];
int top = -1;
void push(int item);
int pop();
int peek();
int isEmpty();
int isFull();
void display();
```

```
main()
{
        int choice,item;
        while(1)
        {
                printf("1.Push\n");
                printf("2.Pop\n");
                printf("3.Display the top element\n");
                printf("4.Display all stack elements\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1 :
                        printf("Enter the item to be pushed : ");
                        scanf("%d",&item);
                        push(item);
                        break;
                 case 2:
                        item = pop();
                        printf("Popped item is : %d\n",item);
                        break;
                 case 3:
                        printf("Item at the top is : %d\n", peek());
                        break;
                 case 4:
                        display();
                        break;
                 case 5:
                        exit(1);
                 default:
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/

void push(int item)
{
        if(isFull())
        {
                printf("Stack Overflow\n");
                return;
        }
        top = top+1;
        stack_arr[top] = item;
}/*End of push()*/

int pop()
{
        int item;
        if(isEmpty())
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        item = stack_arr[top];
        top = top-1;
        return item;
}/*End of pop()*/

int peek()
{
        if(isEmpty())
        {
                printf("Stack Underflow\n");
```

```
                exit(1);
        }
        return stack_arr[top];
}/*End of peek()*/

int isEmpty()
{
        if(top==-1)
                return 1;
        else
                return 0;
}/*End of isEmpty()*/

int isFull()
{
        if(top==MAX-1)
                return 1;
        else
                return 0;
}/*End of isFull()*/

void display()
{
        int i;
        if(isEmpty())
        {
                printf("Stack is empty\n");
                return;
        }
        printf("Stack elements :\n\n");
        for(i=top; i>=0; i--)
                printf(" %d\n", stack_arr[i]);
        printf("\n");
}/*End of display()*/
```

The function `push()` pushes an item on the stack, the function `pop()` pops an item from the stack and returns the popped item to `main()`. The function `peek()` returns top item without removing it from the stack, so the value of `top` remains unchanged. The function `display()` displays all the elements of the stack.

4.1.2 Linked List Implementation of Stack

When the size of stack is not known in advance, it is better to implement it as a linked list. In this case stack will not overflow till there is space available for dynamic memory allocation. We will take a single linked list so the structure of node would be-

```
struct node{
            int info;
            struct node *link;
          };
```

We will take the beginning of linked list as the top of the stack. For push operation, a node will be inserted in the beginning of the list. For pop operation, first node of the list will be deleted. If we take the end of the list as top of the stack then for each push and pop operation we will have to traverse the whole list. We will take a pointer `top` that points to the first node of linked list. This pointer `top` is same as the pointer `start` that we had taken in single linked list.

For pushing an element on stack we follow the procedure of insertion in beginning of the linked list. The function `push()` would be similar to the function `addatbeg()` of single linked list. The stack will overflow only when there is no space left for dynamic memory allocation and in this case call to function `malloc()` will return NULL. So inside the function `push()`, we will check for this overflow condition.

For pop operation, we will delete the first element of linked list. The underflow condition will arise when linked list is empty i.e. when top is equal to NULL. So inside the function pop(), we will check for this underflow condition.

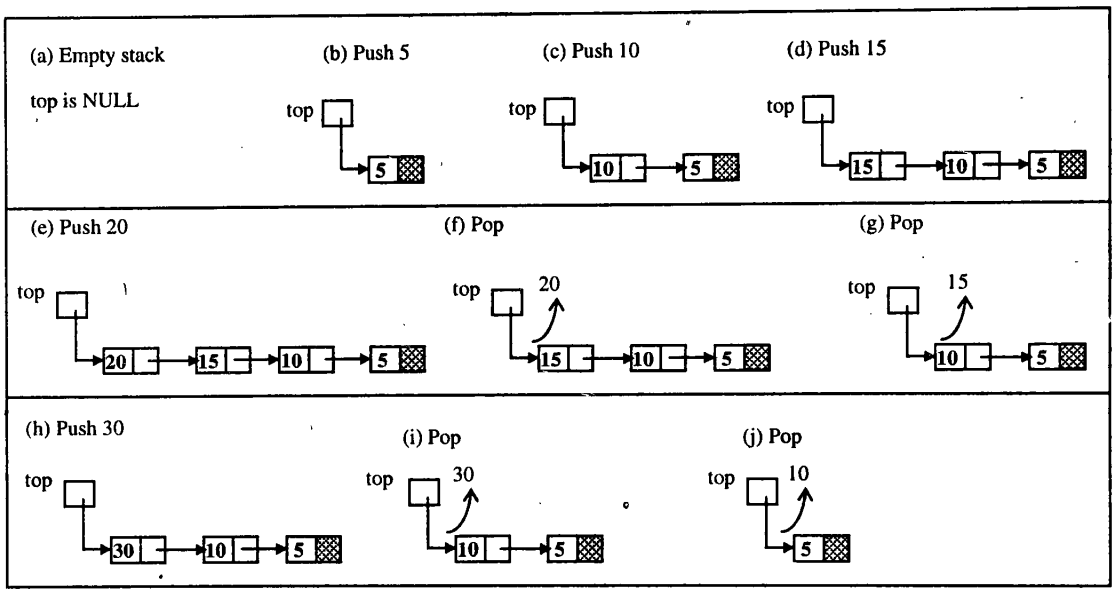


Figure 4.4

```
/*P4.2 Program of stack using linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
}*top = NULL;
void push(int item);
int pop();
int peek();
int isEmpty();
void display();

main()
{
        int choice,item;
        while(1)
        {
                printf("1.Push\n");
                printf("2.Pop\n");
                printf("3.Display item at the top\n");
                printf("4.Display all items of the stack\n");
                printf("5.Quit\n");
                printf("Enter your choice : ") ;
                scanf("%d", &choice);
                switch(choice)
                {
                case 1:
                        printf("Enter the item to be pushed : ");
                        scanf("%d",&item);
                        push(item);
                        break;
                case 2:
```

```
                        item = pop();
                        printf("Popped item is : %d\n",item);
                        break;
                case 3:
                        printf("Item at the top is %d\n",peek());
                        break;
                case 4:
                        display();
                        break;
                case 5:
                        exit(1);
                default :
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/

void push(int item)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        if(tmp==NULL)
        {
                printf("Stack Overflow\n");
                return;
        }
        tmp->info = item;
        tmp->link = top;
        top = tmp;
}/*End of push()*/

int pop()
{
        struct node *tmp;
        int item;
        if(isEmpty())
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        tmp = top;
        item = tmp->info;
        top = top->link;
        free(tmp);
        return item;
}/*End of pop()*/

int peek()
{
        if(isEmpty())
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        return top->info;
}/*End of peek()*/

int isEmpty()
{
        if(top==NULL)
                return 1;
        else
                return 0;
}/*isEmpty()*/
```

```
void display()
{
        struct node *ptr;
        ptr = top;
        if(isEmpty())
        {
                printf("Stack is empty\n");
                return;
        }
        printf("Stack elements :\n");
        while(ptr!=NULL)
        {
                printf(" %d\n",ptr->info);
                ptr = ptr->link;
        }
        printf("\n");
}/*End of display()*/
```

4.2 Queue

Queue is a linear list in which elements can be inserted only at one end called rear of the queue and deleted only at the other end called front of the queue. We can see examples of queue in daily life like queue of people waiting at a counter or a queue of cars etc. In the queue of people and queue of cars, the person or car that enters first in the queue will be out first. The behaviour of queue is first in first out, so it is also called FIFO(First In First Out) data structure. The following example shows that the new element is inserted at the end called `rear` and the deletion is done at the other end called `front`.

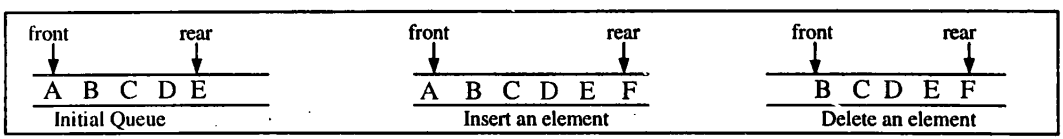


Figure 4.5

In a queue, the insertion operation is known as **enqueue** and deletion is known as **dequeue**. If insert operation is attempted and there is not enough space in the queue, then this situation is called overflow and the new element can't be inserted. If queue is empty and delete operation is attempted, then this situation is called underflow.

4.2.1 Array Implementation of Queue

In stack, both operations were performed at the same end, so we had to take only one variable `top`, but here operations are performed at different ends so we have to take two variables to keep track of both the ends. We will take the variables named `rear` and `front`, where `rear` will hold the index of last added item in queue and `front` will hold the index of first item of queue.

(i) Initially when the queue is empty, the values of both `front` and `rear` will be -1.
(ii) For insertion, the value of `rear` is incremented by 1 and the element is inserted at the new rear position.
(iii) For deletion, the element at front position is deleted and the value of `front` is incremented by 1.
(iv) When insertion is done in an initially empty queue, i.e. if the value of `front` is -1, then value of `front` is made 0.

The following figure shows insertions and deletion in a queue.

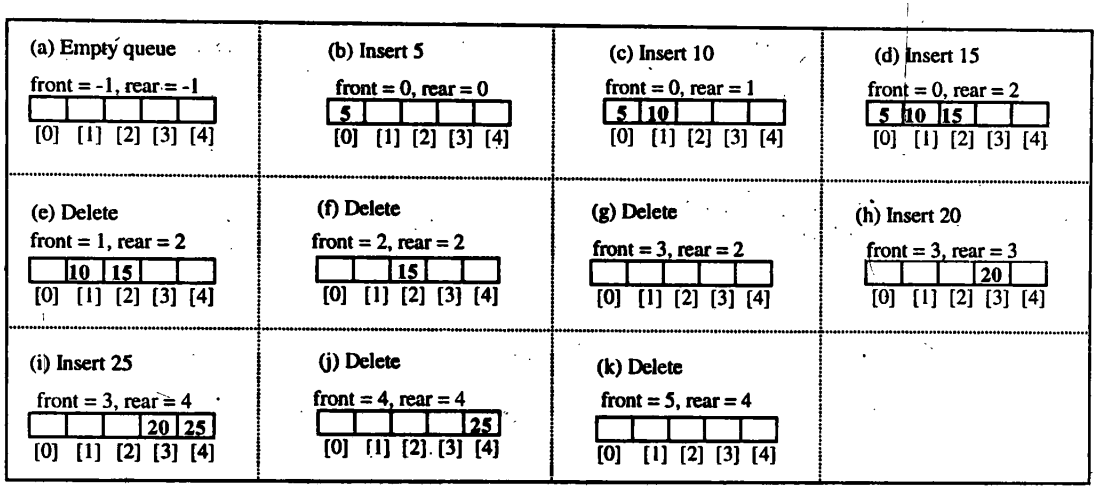

(a) Empty queue
front = -1, rear = -1
[0] [1] [2] [3] [4]
(b) Insert 5
front = 0, rear = 0
5
[0] [1] [2] [3] [4]
(c) Insert 10
front = 0, rear = 1
5 10
[0] [1] [2] [3] [4]
(d) Insert 15
front = 0, rear = 2
5 10 15
[0] [1] [2] [3] [4]
(e) Delete
front = 1, rear = 2
10 15
[0] [1] [2] [3] [4]
(f) Delete
front = 2, rear = 2
15
[0] [1] [2] [3] [4]
(g) Delete
front = 3, rear = 2
[0] [1] [2] [3] [4]
(h) Insert 20
front = 3, rear = 3
20
[0] [1] [2] [3] [4]
(i) Insert 25
front = 3, rear = 4
20 25
[0] [1] [2] [3] [4]
(j) Delete
front = 4, rear = 4
25
[0] [1] [2] [3] [4]
(k) Delete
front = 5, rear = 4
[0] [1] [2] [3] [4]

Figure 4.6

From the figure 4.6, we can note the following things-

(i) At any time the number of elements in the queue is equal to `(rear-front+1)`, except initially empty queue.

(ii) When `front` is equal to `rear`, there is only one element in the queue ((b),(f),(h),(j)), except initially empty queue.

(iii) When `front` becomes equal to `(rear+1)`, the queue becomes empty ((g) and (k)). So we can see that the queue is empty in two situations, when initially `front` is equal to -1 or when `front` becomes equal to `(rear+1)`. These are the two conditions of queue underflow.

(iv) When `rear` becomes equal to 4 it can't be incremented further. After case (i), value of `rear` becomes 4, so now it is not possible to insert any element in the queue. Hence we can say that if the size of array is MAX, then it is not possible to insert elements after `rear` becomes equal to MAX-1. This is the condition for queue overflow.

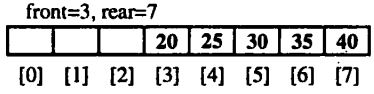
The function `insert()` will insert an item in the queue and the function `del()` will delete an item from the queue. Inside the function `insert()`, first we will check the condition of overflow and then insert the element. Inside the function `del()`, first we will check the condition for underflow and then delete the element. The function `peek()` returns the item at the front of the queue without removing it.

```
/*P4.3 Program of queue using array*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 10
int queue_arr[MAX];
int rear = -1;
int front = -1;
void insert(int item);
int del();
int peek();
void display();
int isFull();
int isEmpty();

main()
{
        int choice,item;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
```

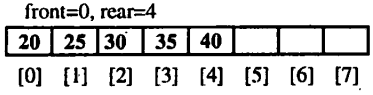
```
                printf("3.Display element at the front\n");
                printf("4.Display all elements of the queue\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                case 1:
                        printf("Input the element for adding in queue : ");
                        scanf("%d",&item);
                        insert(item);
                        break;
                case 2:
                        item = del();
                        printf("Deleted element is %d\n",item);
                        break;
                case 3:
                        printf("Element at the front is %d\n",peek());
                        break;
                case 4:
                        display();
                        break;
                case 5:
                        exit(1);
                default:
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/

void insert(int item)
{
        if(isFull())
        {
                printf("Queue Overflow\n");
                return;
        }
        if(front == -1)
                front = 0;
        rear = rear+1;
        queue_arr[rear] = item ;
}/*End of insert()*/

int del()
{
        int item;
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        item = queue_arr[front];
        front = front+1;
        return item;
}/*End of del()*/

int peek()
{
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        return queue_arr[front];
}/*End of peek()*/
```

```
int isEmpty()
{
        if(front==-1 || front==rear+1)
                return 1;
        else
                return 0;
}/*End of isEmpty()*/

int isFull()
{
        if(rear==MAX-1)
                return 1;
        else
                return 0;
}/*End of isFull()*/

void display()
{
        int i;
        if(isEmpty())
        {
                printf("Queue is empty\n");
                return;
        }
        printf("Queue is :\n\n");
        for(i=front; i<=rear; i++)
                printf("%d  ",queue_arr[i]);
        printf("\n\n");
}/*End of display() */
```

There is a drawback in this array implementation of queue. Consider the situation when `rear` is at the last position of array and `front` is not at the 0^{th} position.



There are 3 spaces for adding the elements but we cannot insert any element in queue because `rear` is at the last position of array. One solution to avoid this wastage of space is that we can shift all the elements of the array to the left and adjust the values of `front` and `rear` accordingly.



This is practically not a good approach because shifting of elements will consume lot of time. Another efficient solution to this problem is circular queue. We will study about it later in this chapter.

4.2.2 Linked List implementation of Queue

Queue can also be implemented through linked list. The structure of a node will be-

```
struct node {
            int info;
            struct node *link;
        };
```

We will take beginning of linked list as `front` and end of linked list as `rear`. So to insert an item in our queue we will add a node at the end of the list and to delete an item from the queue we will delete a node from the beginning of the list. We will maintain two pointers named `front` and `rear`, where `front` will point to the first node of list and `rear` will point to the last node of the list.

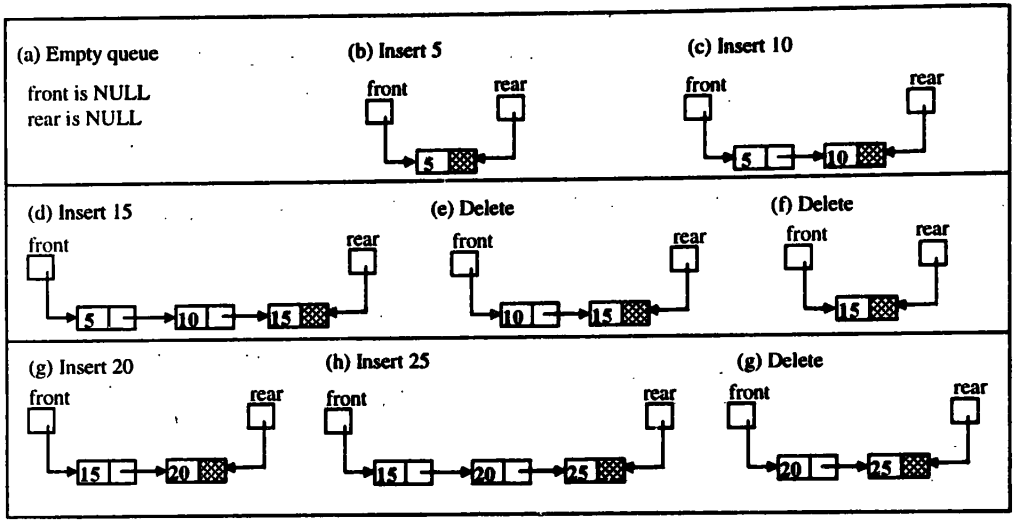


Figure 4.2

```
/* P4.4 Program of queue using linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
}*front = NULL,*rear = NULL;
void insert(int item);
int del();
int peek();
int isEmpty();
void display();

main()
{
        int choice,item;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display the element at the front\n");
                printf("4.Display all elements of the queue\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d", &choice);
                switch(choice)
                {
                case 1:
                        printf("Input the element for adding in queue : ");
                        scanf("%d",&item);
                        insert(item);
                        break;
                case 2:
                        printf("Deleted element is %d\n",del());
                        break;
                case 3:
                        printf("Element at the front of queue is %d\n",peek());
                        break;
                case 4:
                        display();
```

```
                        break;
                case 5:
                        exit(1);
                default :
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/

void insert(int item)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        if(tmp==NULL)
        {
                printf("Memory not available\n");
                return;
        }
        tmp->info = item;
        tmp->link = NULL;
        if(front==NULL)         /*If Queue is empty*/
                front = tmp;
        else
                rear->link = tmp;
        rear = tmp;
}/*End of insert()*/

int del()
{
        struct node *tmp;
        int item;
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        tmp = front;
        item = tmp->info;
        front = front->link;
        free(tmp);
        return item;
}/*End of del()*/

int peek()
{
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        return front->info;
}/*End of peek()*/

int isEmpty()
{
        if(front==NULL)
                return 1;
        else
                return 0;
}/*End of isEmpty()*/

void display()
{
        struct node *ptr;
        ptr = front;
        if(isEmpty())
```

```
        {
                printf("Queue is empty\n");
                return;
        }
        printf("Queue elements :\n\n");
        while(ptr!=NULL)
        {
                printf("%d ",ptr->info);
                ptr = ptr->link;
        }
        printf("\n\n");
}/*End of display()*/
```

We can implement queue with circular linked list also, here we take only one variable `rear`.

```
/*P4.5 Program of queue using circular linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
}*rear = NULL;
void insert(int item);
int del();
void display();
int isEmpty();
int peek();

main()
{
        int choice,item;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Peek\n");
                printf("4.Display\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1:
                        printf("Enter the element for insertion : ");
                        scanf("%d",&item);
                        insert(item);
                        break;
                 case 2:
                        printf("Deleted element is %d\n",del());
                        break;
                 case 3:
                        printf("Item at the front of queue is %d\n",peek());
                        break;
                 case 4:
                        display();
                        break;
                 case 5:
                        exit(1);
                 default:
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
```

```
void insert(int item)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = item;
        if(tmp == NULL)
        {
                printf("Memory not available\n");
                return;
        }
        if(isEmpty()) /*If queue is empty */
        {
                rear = tmp;
                tmp->link = rear;
        }
        else
        {
                tmp->link = rear->link;
                rear->link = tmp;
                rear = tmp;
        }
}/*End of insert()*/

del()
{
        int item;
        struct node *tmp;
        if(isEmpty())
        {
                printf("Queue underflow\n");
                exit(1);
        }
        if(rear->link == rear)  /*If only one element*/
        {
                tmp = rear;
                rear = NULL;
        }
        else
        {
                tmp = rear->link;
                rear->link = rear->link->link;
        }
        item = tmp->info;
        free(tmp);
        return item;
}/*End of del()*/

int peek()
{
        if(isEmpty())
        {
                printf("Queue underflow\n");
                exit(1);
        }
        return rear->link->info;
}/* End of peek() */

int isEmpty()
{
        if(rear==NULL)
                return 1;
        else
                return 0;
}/*End of isEmpty()*/
```

```
void display()
{
        struct node *p;
        if(isEmpty())
        {
                printf("Queue is empty\n");
                return;
        }
        printf("Queue is :\n");
        p = rear->link;
        do
        {
                printf("%d ",p->info);
                p = p->link;
        }while(p!=rear->link);
        printf("\n");
}/*End of display()*/
```

4.3 Circular Queue

We know that when queue is implemented as an array, insertion is not possible after `rear` reaches the last position of array. There may be vacant positions in the array but they can't be utilized. To overcome this limitation we use the concept of circular queue.

We can think of array to be logically circular, so that the two ends of the array wrap up to make a circle.

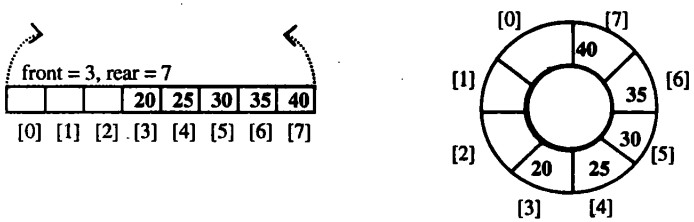


Figure 4.8

Now after the $(n-1)^{th}$ position, 0^{th} position occurs. If we want to insert an element, it can be inserted at 0^{th} position.

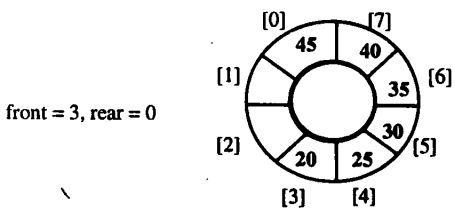


Figure 4.9

The insertion and deletion operations in a circular queue can be performed in a manner similar to that of queue but we have to take care of two things. If value of `rear` is MAX-1, then instead of incrementing `rear` we will make it zero and then perform insertion. Similarly when the value of `front` becomes MAX-1, it will not be incremented but will be reset to zero. Let us take an example and see various operations on a circular queue.

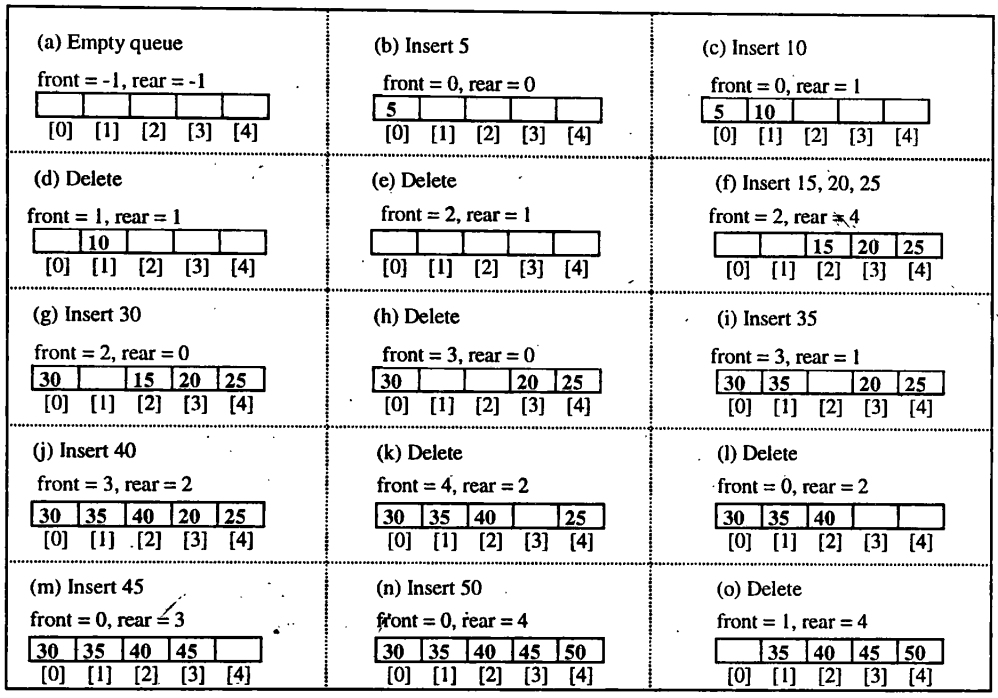

(a) Empty queue
front = -1, rear = -1
[0] [1] [2] [3] [4]
(b) Insert 5
front = 0, rear = 0
5
[0] [1] [2] [3] [4]
(c) Insert 10
front = 0, rear = 1
5 10
[0] [1] [2] [3] [4]
(d) Delete
front = 1, rear = 1
10
[0] [1] [2] [3] [4]
(e) Delete
front = 2, rear = 1
[0] [1] [2] [3] [4]
(f) Insert 15, 20, 25
front = 2, rear = 4
15 20 25
[0] [1] [2] [3] [4]
(g) Insert 30
front = 2, rear = 0
30 15 20 25
[0] [1] [2] [3] [4]
(h) Delete
front = 3, rear = 0
30 20 25
[0] [1] [2] [3] [4]
(i) Insert 35
front = 3, rear = 1
30 35 20 25
[0] [1] [2] [3] [4]
(j) Insert 40
front = 3, rear = 2
30 35 40 20 25
[0] [1] [2] [3] [4]
(k) Delete
front = 4, rear = 2
30 35 40 25
[0] [1] [2] [3] [4]
(l) Delete
front = 0, rear = 2
30 35 40
[0] [1] [2] [3] [4]
(m) Insert 45
front = 0, rear = 3
30 35 40 45
[0] [1] [2] [3] [4]
(n) Insert 50
front = 0, rear = 4
30 35 40 45 50
[0] [1] [2] [3] [4]
(o) Delete
front = 1, rear = 4
35 40 45 50
[0] [1] [2] [3] [4]

Figure 4.10

(i) As in simple queue, here also if `front` is equal to `rear` there is only one element, except initially empty queue (Cases (b) and (d)).

(ii) The circular queue will be empty in three situations, when initially `front` is equal to -1 or when `front` becomes equal to `(rear+1)`, or when `front` is equal to 0 and `rear` is equal to MAX-1 (Cases (a),(e) and the case when all elements deleted from case(o)).

(iii) The overflow condition in circular queue has changed. Here overflow will occur only when all the positions of array are occupied i.e. when the array is full. The array will be full in two situations, when `front` is equal to 0 and `rear` is equal to MAX-1 (Case (n)), or when `front` is equal to `(rear+1)` (Case (j)).

From the last two points we can see that the condition `front==(rear+1)` is true in both cases, when the queue is empty and when the queue is full. Similarly the condition (`front==0 && rear==MAX-1`) is true in both cases. We should make some change in our procedure so that we can differentiate between an empty queue and a full queue.

When the only element of the queue is deleted, `front` and `rear` are reset to -1. We can check for empty queue just by checking the value of `front`, if `front` is -1 then the queue is empty otherwise not. Let us take an example and see how this is done-

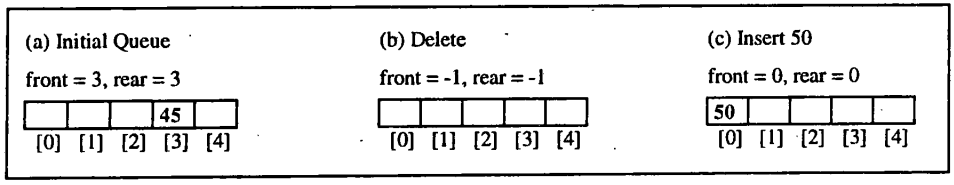


Figure 4.11

The initial queue in figure 4.11 contains only one element. When it is deleted the queue becomes empty, so instead of incrementing `front` we will reset the values of `front` and `rear` to -1. We know that there is only one element in the queue when `front` is equal to `rear`. So inside the function `del()` we can write it as-

```
if(front==rear)
{
      front = rear = -1;
}
```

```
/*P4.6 Program of circular queue*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 10
int cqueue_arr[MAX];
int front = -1;
int rear = -1;
void display();
void insert(item);
int del();
int peek();
int isEmpty();
int isFull();

main()
{
        int choice,item;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Peek\n");
                printf("4.Display\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                case 1 :
                        printf("Input the element for insertion : ");
                        scanf("%d",&item);
                        insert(item);
                        break;
                case 2 :
                        printf("Element deleted is : %d\n",del());
                        break;
                case 3:
                        printf("Element at the front is : %d\n",peek());
                        break;
                case 4:
                        display();
                        break;
                case 5:
                        exit(1);
                default:
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while */
}/*End of main()*/

void insert(int item)
{
        if(isFull())
        {
                printf("Queue Overflow\n");
```

```
                return;
        }
        if(front==-1)
                front = 0;

        if(rear==MAX-1)/*rear is at last position of queue*/
                rear = 0;
        else
                rear = rear+1;
        cqueue_arr[rear] = item ;
}/*End of insert()*/

int del()
{
        int item;
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        item = cqueue_arr[front];
        if(front==rear) /*queue has only one element*/
        {
                front = -1;
                rear = -1;
        }
        else if(front==MAX-1)
                front = 0;
        else
                front = front+1;
        return item;
}/*End of del() */

int isEmpty()
{
        if(front==-1)
                return 1;
        else
                return 0;
}/*End of isEmpty()*/

int isFull()
{
        if((front==0 && rear==MAX-1) || (front==rear+1))
                return 1;
        else
                return 0;
}/*End of isFull()*/

int peek()
{
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        return cqueue_arr[front];
}/*End of peek()*/

void display()
{
        int i;
        if(isEmpty())
        {
                printf("Queue is empty\n");
                return;
```

```
        }
        printf("Queue elements :\n");
        i = front;
        if(front<=rear)
        {
                while(i<=rear)
                        printf("%d ",cqueue_arr[i++]);
        }
        else
        {
                while(i<=MAX-1)
                        printf("%d ",cqueue_arr[i++]);
                i = 0;
                while(i<=rear)
                        printf("%d ",cqueue_arr[i++]);
        }
        printf("\n");
}/*End of display() */
```

4.4 Deque

Deque (pronounced as 'deck' or 'DQ') is a linear list in which elements can be inserted or deleted at either end of the list. The term deque is a short form of **double ended queue**. Like circular queue, we will implement the deque using a circular array(index 0 comes after n-1). For this we have to take an array `deque_arr[]` and two variables `front` and `rear`. The four operations that can be performed on deque are-

(i) Insertion at the front end.
(ii) Insertion at the rear end.
(iii) Deletion from the front end.
(iv) Deletion from the rear end.

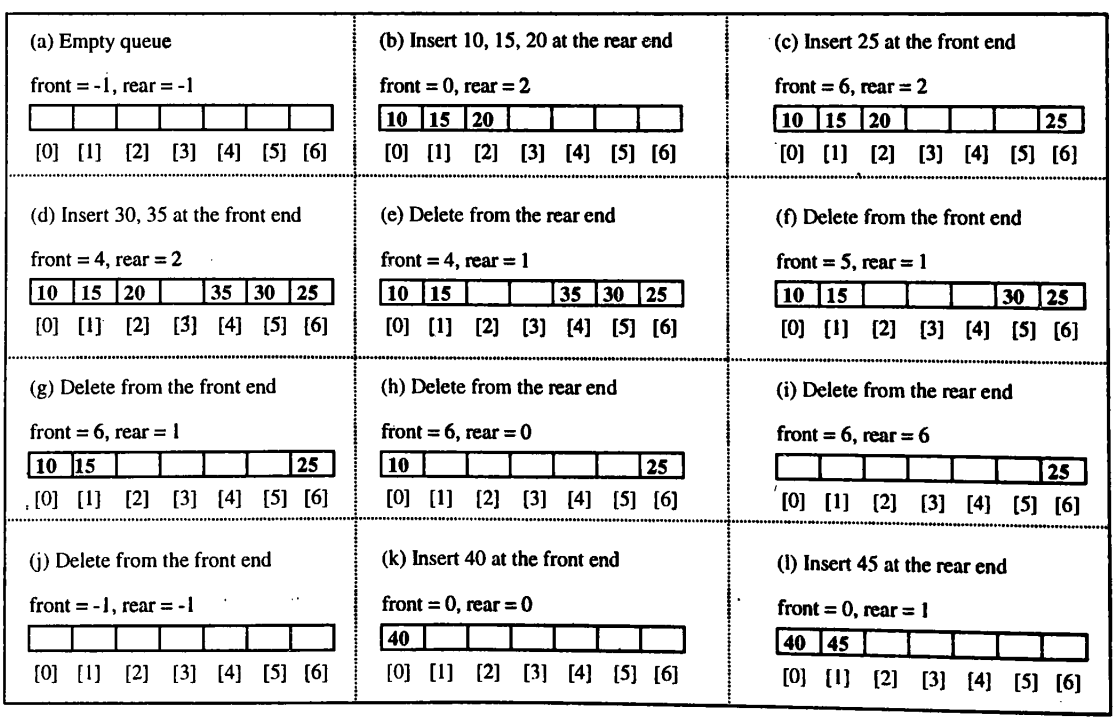

(a) Empty queue
front = -1, rear = -1
[0] [1] [2] [3] [4] [5] [6]
(b) Insert 10, 15, 20 at the rear end
front = 0, rear = 2
10 15 20
[0] [1] [2] [3] [4] [5] [6]
(c) Insert 25 at the front end
front = 6, rear = 2
10 15 20 25
[0] [1] [2] [3] [4] [5] [6]
(d) Insert 30, 35 at the front end
front = 4, rear = 2
10 15 20 35 30 25
[0] [1] [2] [3] [4] [5] [6]
(e) Delete from the rear end
front = 4, rear = 1
10 15 35 30 25
[0] [1] [2] [3] [4] [5] [6]
(f) Delete from the front end
front = 5, rear = 1
10 15 30 25
[0] [1] [2] [3] [4] [5] [6]
(g) Delete from the front end
front = 6, rear = 1
10 15 25
[0] [1] [2] [3] [4] [5] [6]
(h) Delete from the rear end
front = 6, rear = 0
10 25
[0] [1] [2] [3] [4] [5] [6]
(i) Delete from the rear end
front = 6, rear = 6
25
[0] [1] [2] [3] [4] [5] [6]
(j) Delete from the front end
front = -1, rear = -1
[0] [1] [2] [3] [4] [5] [6]
(k) Insert 40 at the front end
front = 0, rear = 0
40
[0] [1] [2] [3] [4] [5] [6]
(l) Insert 45 at the rear end
front = 0, rear = 1
40 45
[0] [1] [2] [3] [4] [5] [6]

Figure 4.12

. Insertion at the rear end and deletion from the front end are performed in similar way as in circular queue. To insert at the front end, the variable `front` is decreased by 1 and then insertion is performed at the new position given by `front`. If the value of `front` is 0 then instead of decrementing, it is made equal to MAX-1. To delete from the rear end, the variable `rear` is decreased by 1. If the value of `rear` is 0, then it is not decremented but it is made equal to MAX-1. The example in figure 4.12 shows various operations on a deque.

```
/*P4.7 Program of deque using circular array*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 7
int deque_arr[MAX];
int front=-1;
int rear=-1;
void insert_frontEnd(int item);
void insert_rearEnd(int item);
int delete_frontEnd();
int delete_rearEnd();
void display();
int isEmpty();
int isFull();

main()
{
        int choice,item;
        while(1)
        {
                printf("1.Insert at the front end\n");
                printf("2.Insert at the rear end\n");
                printf("3.Delete from front end\n");
                printf("4.Delete from rear end\n");
                printf("5.Display\n");
                printf("6.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1:
                        printf("Input the element for adding in queue : ");
                        scanf("%d",&item);
                        insert_frontEnd(item);
                        break;
                 case 2:
                        printf("Input the element for adding in queue : ");
                        scanf("%d",&item);
                        insert_rearEnd(item);
                        break;
                 case 3:
                        printf("Element deleted is : %d\n",delete_frontEnd());
                        break;
                 case 4:
                        printf("Element deleted is : %d\n",delete_rearEnd());
                        break;
                 case 5:
                        display();
                        break;
                 case 6:
                        exit(1);
                 default:
                        printf("Wrong choice\n");
                }/*End of switch*/
                printf("front = %d, rear =%d\n",front,rear);
                display();
```

```
        }/*End of while*/
}/*End of main()*/

void insert_frontEnd(int item)
{
        if(isFull())
        {
                printf("Queue Overflow\n");
                return;
        }
        if(front==-1)/*If queue is initially empty*/
        {
                front=0;
                rear=0;
        }
        else if(front==0)
                front=MAX-1;
        else
                front=front-1;
        deque_arr[front]=item ;
}/*End of insert_frontEnd()*/

void insert_rearEnd(int item)
{
        if(isFull())
        {
                printf("Queue Overflow\n");
                return;
        }
        if(front==-1)  /*if queue is initially empty*/
        {
                front=0;
                rear=0;
        }
        else if(rear==MAX-1)  /*rear is at last position of queue*/
                rear=0;
        else
                rear=rear+1;
        deque_arr[rear]=item ;
}/*End of insert_rearEnd()*/

int delete_frontEnd()
{
        int item;
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        item=deque_arr[front];
        if(front==rear) /*Queue has only one element*/
        {
                front=-1;
                rear=-1;
        }
        else
                if(front==MAX-1)
                        front=0;
                else
                        front=front+1;
        return item;
}/*End of delete_frontEnd()*/

int delete_rearEnd()
{
        int item;
```

```
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        item=deque_arr[rear];

        if(front==rear) /*queue has only one element*/
        {
                front=-1;
                rear=-1;
        }
        else if(rear==0)
                rear=MAX-1;
        else
                rear=rear-1;
        return item;
}/*End of delete_rearEnd()*/

int isFull()
{
        if((front==0 && rear==MAX-1) || (front==rear+1))
                return 1;
        else
                return 0;
}/*End of isFull()*/

int isEmpty()
{
        if(front == -1)
                return 1;
        else
                return 0;
}/*End of isEmpty()*/

void display()
{
        int i;
        if(isEmpty())
        {
                printf("Queue is empty\n");
                return;
        }
        printf("Queue elements :\n");
        i=front;
        if(front<=rear)
        {
                while(i<=rear)
                        printf("%d ",deque_arr[i++]);
        }
        else
        {
                while(i<=MAX-1)
                        printf("%d ",deque_arr[i++]);
                i=0;
                while(i<=rear)
                        printf("%d ",deque_arr[i++]);
        }
        printf("\n");
}/*End of display()*/
```

The four operations of deque are named as push(insert at front end), pop(delete from front end), inject(insert at rear end), eject(delete from rear end). Deque can be of two types-

1. Input restricted deque
2. Output restricted deque

In Input restricted deque, elements can be inserted at only one end but deletion can be performed from both ends. The functions valid in this case are `insert_rearEnd()`, `delete_frontEnd()`, `delete_rearEnd()`. In Output restricted deque, elements can be inserted from both ends but deletion is allowed only at one end. The functions valid in this case are `insert_frontEnd()`, `insert_rearEnd()`, `del_frontEnd()`.

4.5 Priority Queue

We have seen earlier that queue is a list of elements in which we can add the element only at one end called rear of the queue and delete the element only at the other end called front of the queue. In priority queue every element of queue has some priority and it is processed based on this priority. An element with higher priority will be processed before the element which has less priority. If two elements have same priority then in this case FIFO rule will follow, i.e. the element which comes first in the queue will be processed first. An example of priority queue is in CPU scheduling algorithm, in which CPU needs to process those jobs first which have higher priority.

There are two ways of implementing a priority queue-

(i) Through queue : In this case insertion is simple because the element is simply added at the rear end as usual. For performing deletion, first the element with highest priority is searched and then deleted.

(ii) Through sorted list : In this case insertion is costly because the element is inserted at the proper place in the list based on its priority. Here deletion is easy since the element with highest priority will always be in the beginning of the list.

In first case insertion is rapid but deletion is O(n) and in second case deletion is rapid but insertion is O(n). If the priority queue is implemented using arrays then in both the above cases, shifting of elements will be required. So it is advantageous to use a linked list, because insertion or deletion in between the linked list is more efficient. We will implement priority queue as a sorted linked list. The structure of the node would be-

```
struct node{
            int priority;
            int info;
            struct node *link;
};
```

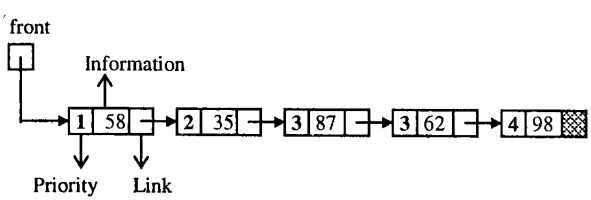
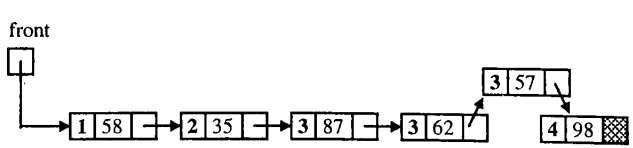


Figure 4.13

Here priority number 1 means the highest priority. If priority number of an element is 2 then it means it has priority more than the element which has priority number 3. Insertion of an element would be performed in a similar way as in sorted linked list. Here we insert the new element on the basis of priority of element.



Delete operation will be the deletion of first element of list because it has more priority than other elements of queue.

```
/*P4.8 Program of priority queue using linked list*/
#include<stdio.h>
#include<stdlib.h>
```

```
struct node
{
        int priority;
        int info;
        struct node *link;
}*front = NULL;
void insert(int item, int item_priority);
int del();
void display();
int isEmpty();

main()
{
        int choice,item,item_priority;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display\n");
                printf("4.Quit\n");
                printf("Enter your choice : ");
                scanf("%d", &choice);
                switch(choice)
                {
                 case 1:
                        printf("Input the item to be added in the queue : ");
                        scanf("%d",&item);
                        printf("Enter its priority : ");
                        scanf("%d",&item_priority);
                        insert(item,item_priority);
                        break;
                 case 2:
                        printf("Deleted item is %d\n",del());
                        break;
                 case 3:
                        display();
                        break;
                 case 4:
                        exit(1);
                 default :
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
void insert(int item,int item_priority)
{
        struct node *tmp,*p;
        tmp = (struct node *)malloc(sizeof(struct node));
        if(tmp == NULL)
        {
                printf("Memory not available\n");
                return;
        }
        tmp->info = item;
        tmp->priority = item_priority;
        /*Queue is empty or item to be added has priority more than first
        element*/
        if(isEmpty() || item_priority < front->priority)
        {
                tmp->link = front;
                front = tmp;
        }
        else
        {
                p = front;
```

```
                while(p->link!=NULL && p->link->priority<=item_priority)
                        p = p->link;
                tmp->link = p->link;
                p->link = tmp;
        }
}/*End of insert()*/
int del()
{
        struct node *tmp;
        int item;
        if(isEmpty())
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        else
        {
                tmp = front;
                item = tmp->info;
                front = front->link;
                free(tmp);
        }
        return item;
}/*End of del()*/
int isEmpty()
{
        if(front==NULL)
                return 1;
        else
                return 0;

}/*End of isEmpty()*/
void display()
{
        struct node *ptr;
        ptr = front;
        if(isEmpty())
                printf("Queue is empty\n");
        else
        {
                printf("Queue is :\n");
                printf("Priority .      Item\n");
                while(ptr!=NULL)
                {
                        printf("%5d        %5d\n",ptr->priority,ptr->info);
                        ptr = ptr->link;
                }
        }
}/*End of display() */
```

The priority queue that we have used is max-priority queue or descending priority queue. The other priority queue is min-priority or ascending priority queue in which the element with lowest priority is processed first. The best implementation of priority queue is through heap tree which we will study in the next chapter.

4.6 Applications of stack

Some of the applications of stack that will be discussed are-

1. Reversal of a string.
2. Checking validity of an expression containing nested parentheses.
3. Function calls.
4. Conversion of infix expression to postfix form.
5. Evaluation of postfix form.

4.6.1 Reversal of string

We can reverse a string by pushing each character of the string on the stack. After the whole string is pushed on the stack, we can start popping the characters from the stack and get the reversed string.

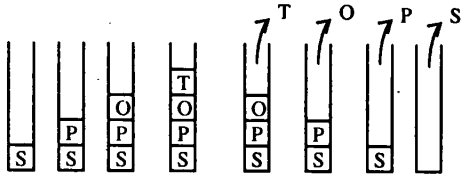


Figure 4.14

We have pushed the string "SPOT" on the stack and we get the reversed string "TOPS".

```
/*P4.9 Program of reversing a string using stack*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 20
int top = -1;
char stack[MAX];
char pop();
void push(char);
main()
{
        char str[20];
        unsigned int i;
        printf("Enter the string : " );
        gets(str);
        /*Push characters of the string str on the stack*/
        for(i=0; i<strlen(str); i++)
                push(str[i]);
        /*Pop characters from the stack and store in string str*/
        for(i=0; i<strlen(str); i++)
                str[i] = pop();
        printf("Reversed string is : ");
        puts(str);
}/*End of main()*/
void push(char item)
{
        if(top==(MAX-1))
        {
                printf("Stack Overflow\n");
                return;
        }
        stack[++top] = item;
}/*End of push()*/
char pop()
{
        if(top==-1)
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        return stack[top--];
}/*End of pop()*/
```

4.6.2 Checking validity of an expression containing nested parentheses

We can use stack to check the validity of an expression that uses nested parentheses. An expression will b valid if it satisfies these two conditions-

1. The total number of left parentheses should be equal to the total number of right parentheses in th expression.
2. For every right parenthesis there should be a left parenthesis of the same type.

Some valid and invalid expressions are given below-

```
[A-B*(C+D)          Invalid
(1+5}               Invalid
[5+4*(9-2)]         Valid
[A+B -(C%D}]        Invalid
[A/(B-C)*D]         Valid
```

The procedure for checking validity of an expression containing nested parentheses is -

1. Initially take an empty stack.
2. Scan the symbols of expression from left to right.
3. If the symbol is a left parenthesis then push it on the stack.
4. If the symbol is right parenthesis
   If the stack is empty
      Invalid : Right parentheses are more than left parentheses.
   else
      Pop an element from stack.
      If popped parenthesis does not match the parenthesis being scanned
         Invalid : Mismatched parentheses.
5. After scanning all the symbols

   If stack is empty
      Valid : Balanced Parentheses.
   else
      Invalid : Left parentheses more than right parentheses.

Some examples are given below-

[A+B-(C%D)]

Symbol	Stack
[	[
A	[
+	[
B	[
-	[
(	[(
C	[(
%	[(
D	[(
)	[

Invalid: Scanned symbol '}' and popped symbol '(' don't match

A-[B*(C+D)

Symbol	Stack
A	Empty
-	Empty
[	[
B	[
*	[
(	[(
C	[(
+	[(
D	[(
)	[

Invalid : All symbols scanned but stack is not empty

[A/(B-C)*D]

Symbol	Stack
[	[
A	[
/	[
(	[(
B	[(
-	[(
C	[(
)	[
*	[
D	[
]	Empty

Valid

A*(B-C)}+D

Symbol	Stack
A	Empty
*	Empty
(	(
B	(
-	(
C	(
)	Empty
}	Empty

Invalid: Scanned symbol ')' but stack is empty

```
/*P4.10 Program to check nesting of parentheses using stack*/
#include<stdio.h>
#define MAX 30
int top = -1;
int stack[MAX];
void push(char);
char pop();
int match(char a,char b);

main()
{
```

```
        char exp[MAX];
        int valid;
        printf("Enter an algebraic expression : ");
        gets(exp);
        valid = check(exp);
        if(valid==1)
                printf("Valid expression\n");
        else
                printf("Invalid expression\n");
}

int check(char exp[])
{
        int i;
        char temp;
        for(i=0; i<strlen(exp); i++)
        {
                if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
                        push(exp[i]);

                if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
                        if(top==-1)     /*stack empty*/
                        {
                                printf("Right parentheses are more than left\n");
                                return 0;
                        }
                        else
                        {
                                temp = pop();
                                if(!match(temp, exp[i]))
                                {
                                        printf("Mismatched parentheses are : ");
                                        printf("%c and %c\n",temp,exp[i]);
                                        return 0;
                                }
                        }
        }
        if(top==-1) /*stack empty*/
        {
                printf("Balanced Parentheses\n");
                return 1;
        }
        else
        {
                printf("Left parentheses more than right parentheses\n");
                return 0;
        }
}/*End of main()*/

int match(char a,char b)
{
        if(a=='[' && b==']')
                return 1;
        if(a=='{' && b=='}')
                return 1;
        if(a=='(' && b==')')
                return 1;
        return 0;
}/*End of match()*/

void push(char item)
{
        if(top==(MAX-1))
        {
                printf("Stack Overflow\n");
```

```
                return;
        }
        top = top+1;
        stack[top] = item;
}/*End of push()*/
char pop()
{
        if(top==-1)
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        return(stack[top--]);
}/*End of pop()*/
```

4.6.3 Function calls

The function calls behave in LIFO manner, i.e. the function that is called first is the last one to finish execution or we can say that the function calls return in the reverse order of their invocation. So stack is the perfect data structure to implement function calls.

A stack is maintained during the execution of the program called program stack or run-time stack. Whenever a function is called the program stores all the information associated with this call in a structure called activation record and this activation record is pushed on the run-time stack. Activation record also known as stack frame consists of the following data-

(i) Parameters and local variables.
(ii) Pointer to previous activation record i.e. activation record of the caller.
(iii) Return address i.e. the instruction in the caller function which is immediately after the function call.
(iv) Return value if function is not void.

The function whose activation record is at the top of the stack is the one that is currently being executed. Whenever a function is called, its activation record is pushed on the stack. When a function terminates its activation record is popped from the stack and after this the activation record of its caller comes at the top. Let us take a C program example in which `main()` calls `f1()`, `f1()` calls `f2()` and `f2()` calls `f3()`. The different stages of run-time stack in this case are shown in figure 4.15. AR in the figure denotes activation record.

If a function calls itself, then the program creates different activation records for different calls of the same function(recursive calls). So we can see that there is no need of any special procedure to implement recursion.

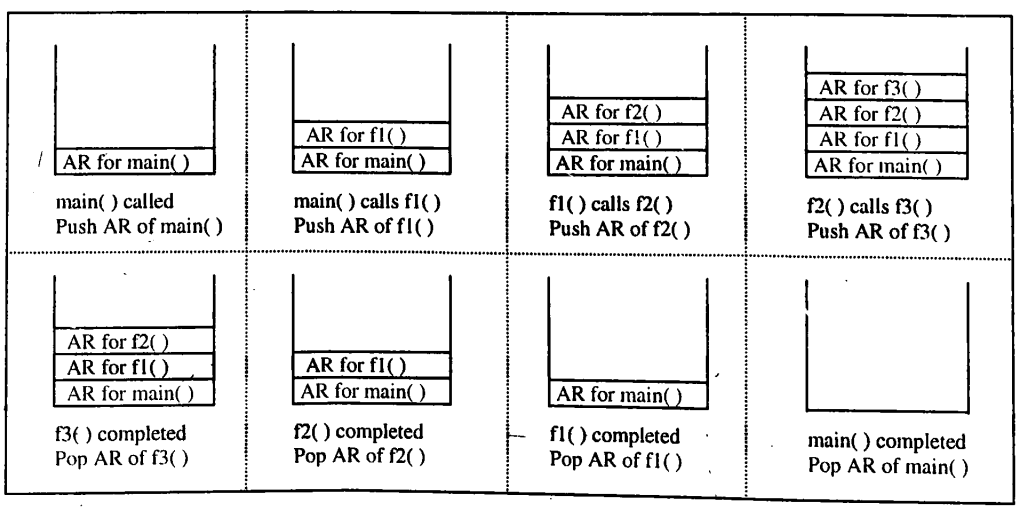

AR for main( )
main( ) called
Push AR of main( )
AR for f1( )
AR for main( )
main( ) calls f1( )
Push AR of f1( )
AR for f2( )
AR for f1( )
AR for main( )
f1( ) calls f2( )
Push AR of f2( )
AR for f3( )
AR for f2( )
AR for f1( )
AR for main( )
f2( ) calls f3( )
Push AR of f3( )
AR for f2( )
AR for f1( )
AR for main( )
f3( ) completed
Pop AR of f3( )
AR for f1( )
AR for main( )
f2( ) completed
Pop AR of f2( )
AR for main( )
f1( ) completed
Pop AR of f1( )
main( ) completed
Pop AR of main( )

Figure 4.15

4.6.4 Polish Notation

An important application of stack is in evaluating arithmetic expressions. An arithmetic expression consists of operators and operands where operands may be either numeric constants or numeric variables. Let us take an arithmetic expression-

9+6/3

If division is performed before addition then the result would be 11, while if addition is performed before division then the result would be 5.

To avoid this ambiguity, we can use parentheses to specify which operation is to be performed first.

(9+6)/3 or 9+(6/3)

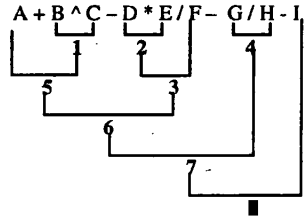
Another solution is that each operation is given some priority, and the operation which has highest priority is performed first.

In our discussion we will take five operators '+', '-', '*', '/' and '^'. The priorities of these operators would be-

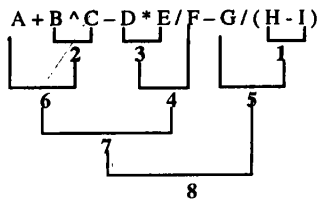
Operator	Priority
^ (Exponentiation)	3
* (Multiplication) and / (Division)	2
+ (Addition) and - (Subtraction)	1

So '^' has highest priority and '+', '-' have lowest priority. If two operators have same priority(like '*' and '/') then the expression is scanned from left to right and whichever operator comes first will be evaluated first. This property is called associativity and we've assumed left to right associativity for all our operators.

Now if we apply these precedence rules to our expression 9+6/3, then we see that division should be performed before addition. Let us take an arithmetic expression and see how it can be evaluated using our precedence rules.



The numbers indicate the sequence of evaluation of operators. If we want to override the precedence rules then we can use parentheses. Anything that is between parentheses will be evaluated first. For example if in the above expression we want to evaluate H-I before G/H then we can enclose H-I within parentheses.



Here we can see that the expression inside the parentheses is evaluated first and after that, evaluation is on the basis of operator precedence.

This is how we can evaluate arithmetic expressions manually. We have to know the precedence rules and take care of parentheses. The expression has to be scanned many times and every time we have to reach different place in expression for evaluation.

Now let us see how computer can evaluate these expressions. If we use the same procedure then there will be repeated scanning from left to right which is inefficient. It would be nice if we could transform the above

expression in some form which does not have parentheses and all operators are arranged in proper order according to their precedence. In that case we could evaluate the expression by just scanning the new transformed expression once from left to right. The Polish notations are used for this purpose so let us see what these notations are.

The great Polish mathematician Jan Lukasiewicz had given a technique for representation of arithmetic expressions in early 1920's. According to the two notations that he gave, the operator can be placed before or after operands. Although these notations were not very readable for humans, but proved very useful for compiler designers in generating machine language code for evaluating arithmetic expressions.

The conventional method of representing an arithmetic expression is known as infix because the operator is placed in between the operands. If the operator is placed before the operands then this notation is called prefix or Polish notation. If the operator is placed after the operand then this notation is called postfix or reverse polish notation. So now we have three ways to represent an arithmetic expression-

```
Infix     A+B
Prefix    +AB
Postfix    AB+
```

Let us see how we can convert an infix expression into its equivalent postfix expression. The rules of parentheses and precedence would remain the same. If there are parentheses in the expression, then the portion inside the parentheses will be converted first. After this the operations are converted according to their precedence. After we've converted a particular operation to its postfix, it will be considered as a single operand. We will use square brackets to represent these types of intermediate operands. If there are operators of equal precedence then the operator which is on the left is converted first. Let us take an example-

```
A-B^C+D*E/(F+G)
A-B^C+D*E/[FG+]          Convert F+G to FG+
A-[BC^]+D*E/[FG+]        Convert B^C to BC^
A-[BC^]+[DE*]/[FG+]      Convert D*E to DE*
A-[BC^]+[DE*FG+/]        Convert [DE*]/[FG+] to DE*FG+/
[ABC^-]+[DE*FG+/]        Convert A-[BC^] to ABC^-
ABC^-DE*FG+/+            Convert [ABC^-]+[DE*FG+/] to ABC^-DE*FG+/+
```

Here are few more examples of conversion from infix to postfix.

```
A+B/C-D*E+F              (A+B)/C*D-E^F            A^B/C*D/E^F*G
A+[BC/]-D*E+F            [AB+]/C*D-E^F            [AB^]/C*D/[EF^]*G
A+[BC/]-[DE*]+F          [AB+]/C*D-[EF^]          [AB^C/]*D/[EF^]*G
[ABC/+]-[DE*]+F          [AB+C/]*D-[EF^]          [AB^C/D*]/[EF^]*G
[ABC/+DE*-]+F            [AB+C/D*]-[EF^]          [AB^C/D*EF^/]*G
ABC/+DE*-F+              AB+C/D*EF^-              AB^C/D*EF^/G*

A-B*C+(D-E)*F/G/H        A+B/C^D-E*F/(G+H)        A+B/(C*D^E)-F*G^H
A-B*C+[D-]*F/G/H         A+B/C^D-E*F/[GH+]        A+B/(C*[DE^])-F*G^H
A-[BC*]+[D-F*]/G/H       A+B/[CD^]-E*F/[GH+]      A+B/[CDE^*]-F*G^H
A-[BC*]+[D-F*G/]/H       A+[BCD^/]-[EF*]/[GH+]    A+B/[CDE^*]-F*[GH^]
A-[BC*]+[D-F*G/H/]       A+[BCD^/]-[EF*GH+/]      A+[BCDE^*/]-[FGH^*]
[ABC*-]+[D-F*G/H/]       [ABCD^/+]-[EF*GH+/]      [ABCDE^*/+]-[FGH^*]
ABC*-DE-F*G/H/+          ABCD^/+EF*GH+/-          ABCDE^*/+FGH^*-
```

The conversion from infix to prefix also uses the same rules, but here the operator is placed before the operands. The following examples show conversion of infix form to prefix form.

```
A+B/C-D*E+F              (A+B)/C*D-E^F            A^B/C*D/E^F*G
A+[/BC]-D*E+F            [+AB]/C*D-E^F            [^AB]/C*D/[^EF]*G
A+[/BC]-[*DE]+F          [+AB]/C*D-[^EF]          [/^ABC]*D/[^EF]*G
[+A/BC]-[*DE]+F          [/+ABC]*D-[^EF]          [*/^ABCD]/[^EF]*G
[-+A/BC*DE]+F            [*/+ABCD]-[^EF]          [/*/^ABCD^EF]*G
+-+A/BC*DEF              -*/+ABCD^EF              */*/^ABCD^EFG
```

```
A-B*C+(D-E)*F/G/H          A+B/C^D-E*F/(G+H)          A+B/(C*D^E)-F*G^H
A-B*C+[-DE]*F/G/H          A+B/C^D-E*F/[+GH]          A+B/(C*[^DE])-F*G^H
A-[*BC]+[-DE]*F/G/H        A+B/[^CD]-E*F/[+GH]        A+B/[*C^DE]-F*G^H
A-[*BC]+[*-DEF]/G/H        A+[/B^CD]-E*F/[+GH]        A+B/[*C^DE]-F*[^GH]
A-[*BC]+[/*-DEFG]/H        A+[/B^CD]-[*EF]/[+GH]      A+[/B*C^DE]-F*[^GH]
A-[*BC]+[//*-DEFGH]        A+[/B^CD]-[/*EF+GH]        A+[/B*C^DE]-[*F^GH]
[-A*BC]+[//*-DEFGH]        [+A/B^CD]-[/*EF+GH]        [+A/B*C^DE]-[*F^GH]
+-A*BC//*-DEFGH            -+A/B^CD/*EF+GH            -+A/B*C^DE*F^GH
```

We can see that the prefix and postfix forms of any expression are not mirror image of each other. In all the three forms the relative positions of the operands remain the same.

In the prefix and postfix forms, parentheses are not needed and the operators and operands are arranged in proper order according to their precedence levels. Now let us take an infix expression and covert it to postfix and then see how this postfix expression can be evaluated. For simplicity we will take numerical constants of single digit only.

```
3+5*(7-4)^2
3+5*[74-]^2
3+5*[74-2^]
3+[574-2^*]
3574-2^*+
```

To evaluate this postfix expression, it is scanned from left to right and as soon as we get an operator we apply it to the last two operands.

```
3574-2^*+      Apply '-' to 7 and 4
35[3]2^*+      Apply '^' to 3 and 2
35[9]*+        Apply '*' to 5 and 9
3[45]+         Apply '+' to 3 and 45
48
```

So we can see that in a postfix expression, the operators are evaluated in the same sequence as they appear in the expression. While evaluating a postfix expression we are not concerned about any precedence rules or parentheses.

Now let us see how we can write a program for the two step process of converting the infix form to postfix form and evaluating the postfix expression. The stack data structure proves helpful here and is used in both the steps.

4.6.4.1 Converting infix expression to postfix expression using stack

We will take the operators '^' , '*' , '/', '+' and '-', and assume the same precedence rules as before. We know that the order of operands is same in both infix and postfix. So the operands can be simply added to postfix. In postfix, the order of operators is to be changed and parentheses are to be eliminated. So a stack will be used for temporary placement of operators and left parentheses.

Let us take a character array named `infix` that has arithmetic expression in infix form and another character array `postfix` that will contain the arithmetic expression in postfix form. The steps involved to convert the infix expression into postfix expression will be-

(1) Scan the symbols of array `infix` one by one from left to right.

(a) If symbol is an operand
Add it to array `postfix`.

(b) If symbol is left parenthesis '('
Push it on the stack.

(c) If symbol is right parenthesis ')'
Pop all the operators from stack upto the first left parenthesis and add these operators to array `postfix`. Discard both left and right parentheses.

(d) If symbol is operator
Pop the operators which have precedence greater than or equal to the precedence of the symbol operator, and add these popped operators to array `postfix`.
Push the scanned symbol operator on the stack. (Assume that left parenthesis has least precedence)

(2) After all the symbols of array `infix` have been scanned, pop all the operators remaining on the stack and add them to the array `postfix`.

Let us take an infix expression and convert it into postfix-

Infix : A+B/C*(D+E)-F				
	Symbol	**Action**	**Stack**	**Postfix**
(1)	A	Add A to postfix	Empty	A
(2)	+	Push on the stack	+	A
(3)	B	Add B to postfix	+	AB
(4)	/	Push on the stack	+/	AB
(5)	C	Add C to postfix	+/	ABC
(6)	*	Pop / and add it to postfix and then push * on the stack	+*	ABC/
(7)	(	Push on the stack	+*(	ABC/
(8)	D	Add D to postfix	+*(	ABC/D
(9)	+	Push on the stack	+*(+	ABC/D
(10)	E	Add E to postfix	+*(+	ABC/DE
(11)	)	Pop + and add to postfix	+*	ABC/DE+
(12)	-	Pop * and +, add them to postfix and then push -	-	ABC/DE+*+
(13)	F	Add F to postfix	-	ABC/DE+*+F
(14)		Pop - and add to postfix	Empty	ABC/DE+*+F-

* In step(2) the operator '+' is simply pushed on the stack, since the stack is empty.
* In step(4), first we will compare the precedence of '/' with that of the top operator of stack which is '+'. Since the precedence of '+' is less than that of '/', we will not pop any operator and simply push '/' on the stack.
* In step(6), we will compare the precedence of '*' with that of the top operator of stack which is '/'. Since the precedence of '*' is equal to that of '/', operator '/' is popped and added to postfix. Now the top operator of stack is '+'. Now we will compare precedence of '*' with that of '+'. Since the precedence of '+' is less than that of '*', now operator '*' is pushed on the stack.
* In step(9), '+' is simply pushed on the stack since we have assumed that the precedence of '(' is least.
* In step (11), we have to pop all operators till left parentheses, so '+' is popped and added to postfix.
* In step(12), precedence of '*' is greater than that of '-', so '*' is popped and added to postfix. Now top operator on stack is '+' and its precedence is equal to that of '-', so it is also popped and added to postfix. After this '-' is pushed on the stack.
* In step(14), all the symbols of the expression have been scanned, so now we have to pop all the operators from the stack and add to postfix. The only operator remaining on stack is '-', so it is popped and added to postfix.

Let us work out a few more examples to fully comprehend the process.

Infix : A^B*C/(D*E-F)		
Symbol	**Stack**	**Postfix**
A	Empty	A
^	^	A
B	^	AB
*	*	AB^
C	*	AB^C
/	/	AB^C*
(	/(	AB^C*
D	/(	AB^C*D
*	/(*	AB^C*D
E	/(*	AB^C*DE
-	/(-	AB^C*DE*
F	/(-	AB^C*DE*F
)	/	AB^C*DE*F-
	Empty	AB^C*DE*F-/

Infix : A-B/C^D*E+F/G		
Symbol	**Stack**	**Postfix**
A	Empty	A
-	-	A
B	-	AB
/	-/	AB
C	-/	ABC
^	-/^	ABC
D	-/^	ABCD
*	-*	ABCD^/
E	-*	ABCD^/E
+	+	ABCD^/E*-
F	+	ABCD^/E*-F
/	+/	ABCD^/E*-F
G	+/	ABCD^/E*-FG
	Empty	ABCD^/E*-FG/+

Infix : 8*(5^4+2)-6^2/(9*3)		
Symbol	**Stack**	**Postfix**
8	Empty	8
*	*	8
(	*(	8
5	*(	85
^	*(^	85
4	*(^	854
+	*(+	854^
2	*(+	854^2
)	*	854^2+
-	-	854^2+*
6	-	854^2+*6
^	-^	854^2+*6
2	-^	854^2+*62
/	-/	854^2+*62^
(	-/(	854^2+*62^
9	-/(	854^2+*62^9
*	-/(*	854^2+*62^9
3	-/(*	854^2+*62^93
)	-/	854^2+*62^93*
	Empty	854^2+*62^93*/-

Infix : 7+5*3^2/(9-2^2)+6*4		
Symbol	**Stack**	**Postfix**
7	Empty	7
+	+	7
5	+	75
*	+*	75
3	+*	753
^	+*^	753
2	+*^	7532
/	+/	7532^*
(	+/(	7532^*
9	+/(	7532^*9
-	+/(-	7532^*9
2	+/(-	7532^*92
^	+/(-^	7532^*92
2	+/(-^	7532^*922
)	+/	7532^*922^-
+	+	7532^*922^-/+
6	+	7532^*922^-/+6
*	+*	7532^*922^-/+6
4	+*	7532^*922^-/+64
	Empty	7532^*922^-/+64*+

4.6.4.2 Evaluation of postfix expression using stack

In a postfix expression operators are placed after the operands. When we scan the postfix expression from left to right, first the operands will come and then the corresponding operator. So this time we will need stack to hold the operands. Whenever any operator occurs in scanning, we pop two operands from stack and perform the operation. Here we do not require any information of operator precedence.

Let us take an array `postfix` that has arithmetic expression in postfix form. The steps involved to evaluate the postfix expression are-

1. Scan the symbols of array postfix one by one from left to right.

   (a) If symbol is operand
   Push it on the stack
   (b) If symbol is operator
   Pop two elements from stack and apply the operator to these two elements
   Suppose first A is popped and then B is popped
   result = B operator A
   Push result on the stack

2. After all the symbols of postfix have been scanned, pop the only element left in the stack and it is the value of postfix arithmetic expression.

Let us take a postfix expression and evaluate it-

Infix : 7+5*3^2/(9-2^2)+6*4 Postfix : 7532^*922^-/+64*+		
Symbol	**Action**	**Stack**
7	Push 7	7
5	Push 5	7,5
3	Push 3	7,5,3
2	Push 2	7,5,3,2
^	Pop 2 and 3, Push 3^2	7,5,9
*	Pop 9 and 5, Push 5*9	7,45
9	Push 9	7,45,9
2	Push 2	7,45,9,2
2	Push 2	7,45,9,2,2
^	Pop 2 and 2, Push 2^2	7,45,9,4
-	Pop 4 and 9, Push 9-4	7,45,5
/	Pop 5 and 45, Push 45/5	7,9
+	Pop 9 and 7, Push 7+9	16
6	Push 6	16,6
4	Push 4	16,6,4
*	Pop 4 and 6, Push 6*4	16,24
+	Pop 24 and 16, Push 16+24	40

After evaluation of the postfix expression its value is 40. Let us take the same postfix expression in infix form and evaluate it.

```
7+5*3^2/(9-2^2)+6*4
7+5*3^2/(9-4)+6*4
7+5*3^2/5+6*4
7+5*9/5+6*4
7+45/5+6*4
7+9+6*4
7+9+24
16+24
40
```

We can see the same result but in postfix evaluation we have no need to take care of parentheses and sequence of evaluation.

```
/* P4.11 Program for conversion of infix to postfix and evaluation of postfix.
It will evaluate only single digit numbers*/
#include<stdio.h>
#include<string.h>
#include<math.h>
#include<stdlib.h>
#define BLANK ' '
#define TAB '\t'
#define MAX 50
void push(long int symbol);
long int pop();
void infix_to_postfix();
long int eval_post();
int priority(char symbol);
int isEmpty();
int white_space();
char infix[MAX],postfix[MAX];
long int stack[MAX];
int top;

main()
{
        long int value;
        top = -1;
        printf("Enter infix : ");
        gets(infix);
        infix_to_postfix();
        printf("Postfix : %s\n",postfix);
        value = eval_post();
```

```
        printf("Value of expression : %ld\n",value);
}/*End of main()*/

void infix_to_postfix()
{
        unsigned int i,p = 0;
        char next;
        char symbol;
        for(i=0; i<strlen(infix); i++)
        {
                symbol = infix[i];
                if(!white_space(symbol))
                {
                   switch(symbol)
                   {
                    case '(':
                        push(symbol);
                        break;
                    case ')':
                        while( (next=pop())!='(' )
                          postfix[p++] = next;
                        break;
                    case '+':
                    case '-':
                    case '*':
                    case '/':
                    case '%':
                    case '^':
                     while(!isEmpty() && priority(stack[top])>=priority(symbol))
                                postfix[p++] = pop();
                        push(symbol);
                        break;
                    default: /*if an operand comes*/
                        postfix[p++] = symbol;
                   }
                }
        }
        while(!isEmpty())
                postfix[p++] = pop();
        postfix[p]='\0'; /*End postfix with '\0' to make it a string*/
}/*End of infix_to_postfix()*/

/*This function returns the priority of the operator*/
int priority(char symbol)
{
        switch(symbol)
        {
        case '(':
                return 0;
        case '+':
        case '-':
                return 1;
        case '*':
        case '/':
        case '%':
                return 2;
        case '^':
                return 3;
        default:
                return 0;
        }
}/*End of priority()*/

void push(long int symbol)
{
        if(top>MAX)
```

```
        {
                printf("Stack overflow\n");
                exit(1);
        }
        stack[++top] = symbol;
}/*End of push()*/

long int pop()
{
        if(isEmpty())
        {
                printf("Stack underflow\n");
                exit(1);
        }
        return (stack[top--]);
}/*End of pop()*/

int isEmpty()
{
        if(top==-1)
                return 1;
        else
                return 0;
}/*End of isEmpty()*/

int white_space(char symbol)
{
        if(symbol==BLANK || symbol==TAB)
                return 1;
        else
                return 0;
}/*End of white_space()*/

long int eval_post()
{
        long int a,b,temp,result;
        unsigned int i;

        for(i=0; i<strlen(postfix); i++)
        {
                if(postfix[i]<='9' && postfix[i]>='0')
                        push( postfix[i]-'0' );
                else
                {
                        a = pop();
                        b = pop();
                        switch(postfix[i])
                        {
                        case '+':
                                temp = b+a; break;
                        case '-':
                                temp = b-a;break;
                        case '*':
                                temp = b*a;break;
                        case '/':
                                temp = b/a;break;
                        case '%':
                                temp = b%a;break;
                        case '^':
                                temp = pow(b,a);
                        }
                        push(temp);
                }
        }
        result = pop();
        return result;
```

```
}/*End of eval_post */
```

In the above discussion we have taken all our operators as left associative for simplicity i.e. when two operators have same precedence then first we evaluated the operator which was on the left. But the exponentiation operation is generally right associative. The value of expression 2^2^3 should be 2^8=256 and not 4^3=64. So A^B^C should be converted to ABC^^, while we are converting it to AB^C^. So let us see what change we can make in our procedure if we have any operator which is right associative. We will define two priorities in this case-

Operator	In-stack Priority	Incoming Priority
^ (Exponentiation)	3	4
* (Multiplication) and / (Division)	2	2
+ (Addition) and - (Subtraction)	1	1

For left associative operators both priorities will be same but for right associative operators incoming priority should be more than the in-stack priority. In our program we will make two functions instead of function `priority()`. Now whenever an operator comes while scanning infix, then pop the operators which have in-stack priority greater than or equal to the incoming priority of the symbol operator.

```
while(!isempty() && instack_pr(stack[top])>=incoming_pr(symbol))
      postfix[p++] = pop();
```

Here `instack_pr()` and `incoming_pr()` are functions that return the in-stack and incoming priorities of operators.

Exercise

1. Implement a stack in an array A, using A[0] as the top.
2. Implement a queue in an array such that if rear reaches the end, all the elements are shifted left.
3. Implement a stack in an array such that the stack starts from the last position of array i.e. initially the top is at the last position of the array, and it moves towards the beginning of array as the elements are pushed.
4. Write a program to implement 2 stacks in a single array of size N. The stacks should not show overflow if there is even a single slot available in the array. Write the functions `popA()`, `popB()`, `pushA()`, `pushB()` for pushing and popping from these stacks.
5. Write a program to implement 2 queues in a single array. The queues should not show overflow if there is even a single slot available in the array.
6. Implement a queue using 2 stacks.
7. Implement a stack using 2 queues.
8. Implement a stack with one queue
9. Reverse a stack using two stacks.
10. Reverse a stack using a queue.
11. Reverse a queue using one stack.
12. Copy the contents of a stack S1 to another stack S2 using one temporary stack
13. Copy a queue Q1 to another queue Q2.
14. Implement a data structure that supports stack push and pop operations and returns the minimum element.
15. Write a program to convert a decimal number to binary using a stack.
16. Write a program to print all the prime factors of a number in descending order using a stack. For example the prime divisors of 450 are 5, 5, 3, 3, 2.
17. Write a program for converting an infix expression to prefix and evaluating that prefix expression.
18. Convert following infix expressions to equivalent postfix and prefix
(i) (A+B)*(C+D) (ii) A%(C-D)+B*E (iii) A-(B+C)*D/E (iv) H^(J+K)*I%S
19. Convert the following infix expressions to equivalent prefix expressions and then evaluate the prefix expressions using stack.

(i) (5+3)/4-2 (ii) 6/3+3*2^2+1 (iii) (8+2)/2 *3-2/1+3

20. Convert the following postfix expressions to prefix-

(i) ABC*-DE-F*G/H/+ (ii) AB+C/D*EF^– (iii) ABC^-DE*FG+/+

Write a program for this conversion.

21. Convert the following prefix expressions to postfix

(i) +-+A/BC*DEF (ii) -+A/B*C^DE*F^GH (iii) -+A/B^CD/*EF+GH

Write a program for this conversion.

22. Evaluate the following postfix expressions

(i) 593/21+*+62/-3+ (ii) 432^+18*22+/-2- (iii) 23^1-42/6*+31+2/-

Recursion

Recursion is a process in which a problem is defined in terms of itself. The problem is solved by repeatedly breaking it into smaller problems, which are similar in nature to the original problem. The smaller problems are solved and their solutions are applied to get the final solution of the original problem. To implement recursion technique in programming, a function should be capable of calling itself. A recursive function is a function that calls itself.

```
main()
{
        .....
        rec();
        .....
}/*End of main()*/
void rec()
{
        .....
        rec();          /*recursive call*/
        .....
}/*End of rec()*/
```

Here the function `rec()` is calling itself inside its own function body, so `rec()` is a recursive function. When `main()` calls `rec()`, the code of `rec()` will be executed and since there is a call to `rec()` inside `rec()`, again `rec()` will be executed. It seems that this process will go on infinitely but in practice, a terminating condition is written inside the recursive function which ends this recursion. This terminating condition is also known as exit condition or the base case. This is the case when function will stop calling itself and will finally start returning.

Recursion proceeds by repeatedly breaking a problem into smaller versions of the same problem, till finally we get the smallest version of the problem which is simple enough to solve. The smallest version of problem can be solved without recursion and this is actually the base case.

5.1 Writing a recursive function

Recursion is actually a way of thinking about problems, so before writing a recursive function for a problem we should be able to define the solution of the problem in terms of a similar type of a smaller problem. The two main steps in writing a recursive function are-

1. Identification of the base case and its solution, i.e. the case where solution can be achieved without recursion. There may be more than one base case.
2. Identification of the general case or the recursive case i.e. the case in which recursive call will be made.

Identifying the base case is very important because without it the function will keep on calling itself resulting in infinite recursion.

We must ensure that each recursive call takes us closer to the base case i.e. the size of problem should be diminished at each recursive call. The recursive calls should be made in such a way that finally we arrive at the

base case. If we don't do so, we will have infinite recursion. So merely defining a base case will not help us avoid infinite recursion, we should implement the function such that the base case is finally reached.

5.2 Flow of control in Recursive functions

Before understanding the flow of control in recursive calls, first let us see how control is transferred in simple function calls.

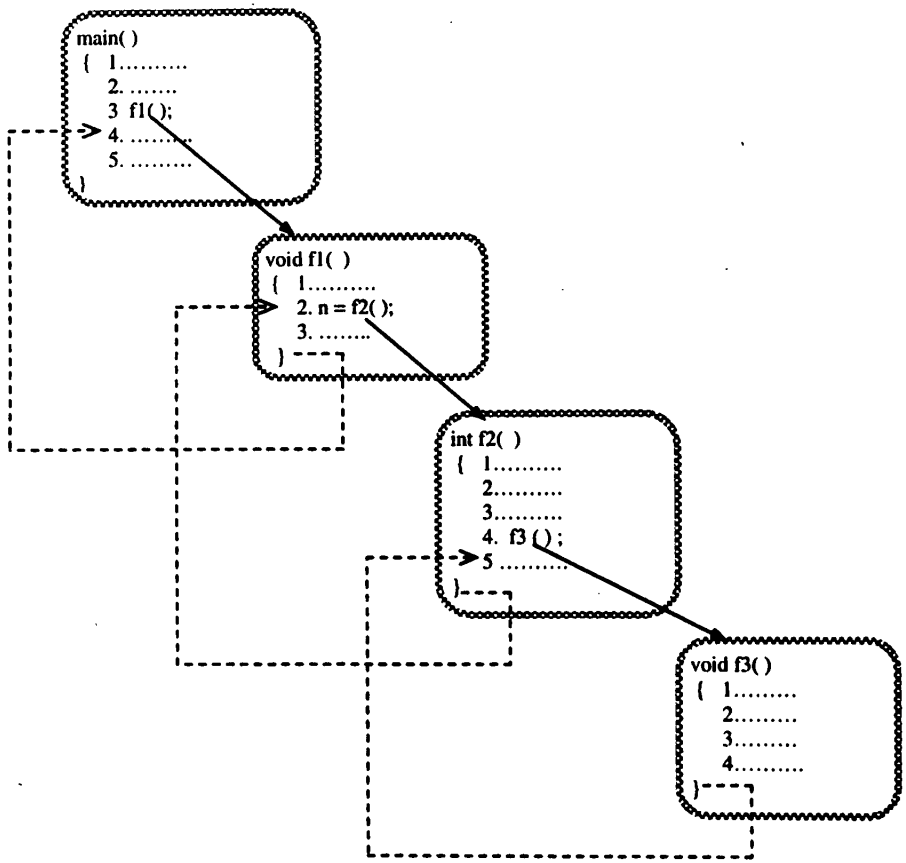


Figure 5.1 Flow of control in normal function calls

In the figure 5.1, the flow of control is-

* `main()` calls function `f1()` at line 3 so control is transferred to `f1()`.
* Inside function `f1()`, the function `f2()` is called at line 2 so control is transferred to function `f2()`.
* Inside function `f2()`, the function `f3()` is called at line 4 so control is transferred to function `f3()`.
* Inside function `f3()` no function is called so control returns back to function `f2()` at line 5.
* After finishing function `f2()`, control is returned to function `f1()` at line 2. Here control returned to line 2 because the work of assigning the return value of `f2()` to variable n still remains.
* Now after finishing the execution of function `f1()` the control is returned to `main()` at line 4.

When the execution of a function is finished we return to the caller(parent) function at the place where we had left it. Recursive calls are no different and they behave in a similar manner. In figure 5.2, the function that is being called each time is the same. We will call the different calls of `func()` as different instances or different invocations of `func()`.

* Initially `main()` calls `func()`, so in first instance of `func()` value of formal parameter n is 5. Line 1 of this function is executed, after this the terminating condition is checked and since it is false we don't return. Now line 4 is executed and then at line 5, `func()` is called with argument 3.

* Now control transfers to the second instance of `func()` and inside this instance, value of n is 3. Line 1 of this second instance of `func()` is executed, after this the terminating condition is checked and since it is false we don't return. Now line 4 is executed and then at line 5, `func()` is called with argument 1.

* Now control transfers to the third instance of `func()` and inside this instance, value of n is 1. Line 1 of this third instance of `func()` is executed, after this the terminating condition is checked and since it is true, we return.

* We return to second instance of `func()` at line 6. Note that now we are inside second instance of `func()` so value of n is 3. After executing lines 6 and 7 we return back to the line 6 of first instance.

* Now we are inside first instance of `func()` so value of n is 5. After executing lines 6 and 7 of first instance we return to `main()` at line 4.

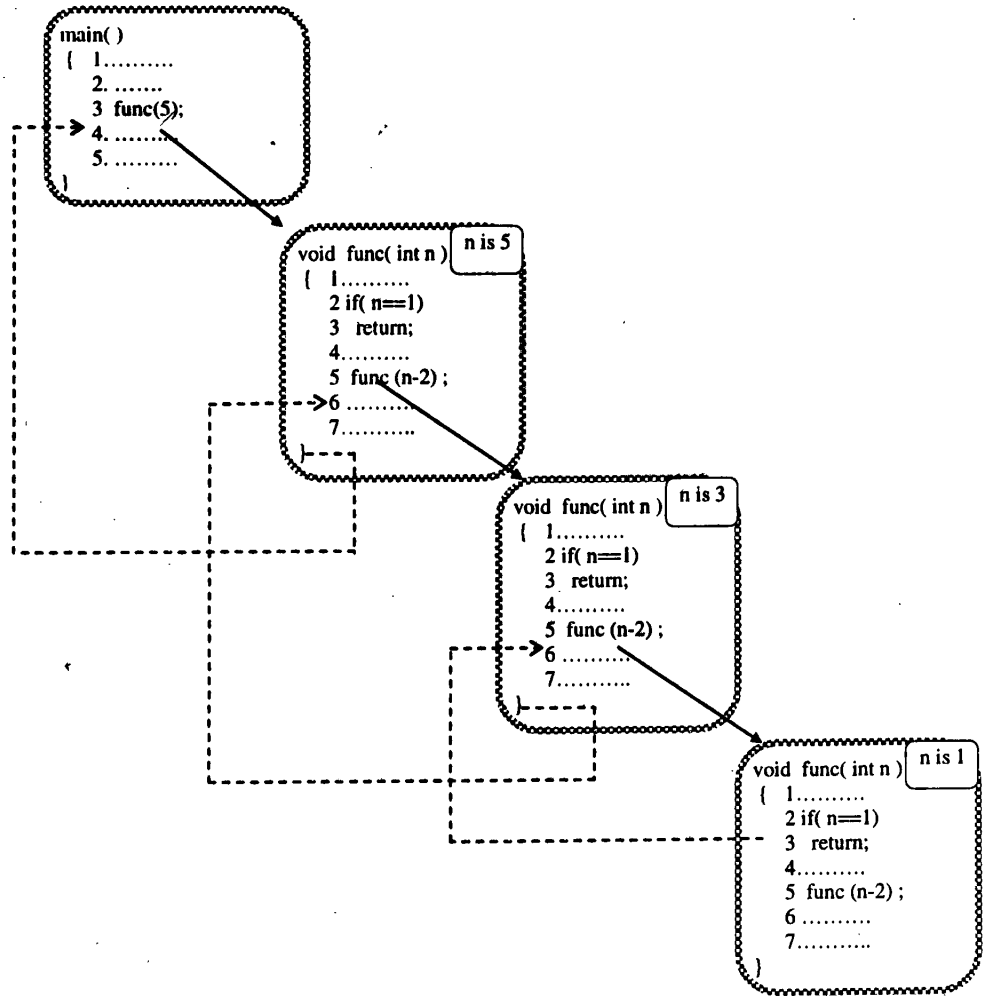


Figure 5.2 Flow of control in recursive function calls

So we can see that the recursive functions are called in a manner similar to that of regular functions, but here the same function is called each time. When execution of an instance of recursive function is finished, we return to the previous instance where we had left it.

We know each function has some local variables that exist only inside that function. The local variables of the called function are active while the local variables of the caller function are kept on hold or suspended. The same case occurs in recursion but here the caller function and called function are copies of the same function. When a function is called recursively, then for each instance a new set of formal parameters and local variables(except static) is created. Their names are same as declared in function but their memory locations are different and they contain different values. These values are remembered by the compiler till the end of function call, so that these values are available while returning. In the example of figure 5.2, we saw that there were three instances of `func()`, but each instance had its own copy of formal parameter n.

The number of times that a function calls itself is known as the recursive depth of that function. In the example of figure 5.2, the depth of recursion is 3.

5.3 Winding and unwinding phase

All recursive functions work in two phases - winding phase and unwinding phase. Winding phase begins when the recursive function is called for the first time, and each recursive call continues the winding phase. In this phase the function keeps on calling itself and no return statements are executed in this phase. This phase terminates when the terminating condition becomes true in a call. After this the unwinding phase begins and all the recursive function calls start returning in reverse order till the first instance of function returns. In unwinding phase the control returns through each instance of the function.

In some algorithms we need to perform some work while returning from recursive calls, in that case we put that particular code in the unwinding phase i.e. just after the recursive call.

We will try to understand the concept of recursion through numerous examples, although most of them may not be very efficient yet they are classic examples of learning how recursion works. Initially when learning recursion, we should trace the functions and see how the control is transferred to understand the behavior of recursive functions.

5.4 Examples of Recursion

5.4.1 Factorial

We know that the factorial of a positive integer n can be found out by multiplying all integers from 1 to n.
n! = 1 * 2 * 3 *.....* (n-1) * n

This is the iterative definition of factorial, and we can easily write an iterative function for it using a loop. Now we will try to find out the solution of this factorial problem recursively.

We know that - 4! = 4 * 3 * 2 * 1
We can write it as- 4! = 4 * 3! (since 3! = 3 * 2 * 1)
Similarly we can write- 3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!

So in general we can say that factorial of a positive integer n is the product of n and factorial of n-1.

n! = n * (n-1)!

Now problem of finding out factorial of (n-1) is similar to that of finding out factorial of n, but it is smaller in size. So we have defined the solution of factorial problem in terms of itself. We know that the factorial of 0 is 1. This can act as the terminating condition or the base case. So the recursive definition of factorial can be written as-

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

Now we will write a program, which finds out the factorial using a recursive function.

```
/*P5.1 Program to find the factorial of a number by recursive method*/
#include<stdio.h>
long int fact(int n);
main()
{
        int num;

        printf("Enter a number : ");
        scanf("%d", &num);
        if(num<0)
                printf("No factorial for negative number\n");
        else
                printf("Factorial of %d is %ld\n", num, fact(num));
}
long int fact(int n)
{
        if(n == 0)
                return(1);
        return(n * fact(n-1));
}/*End of fact()*/
```

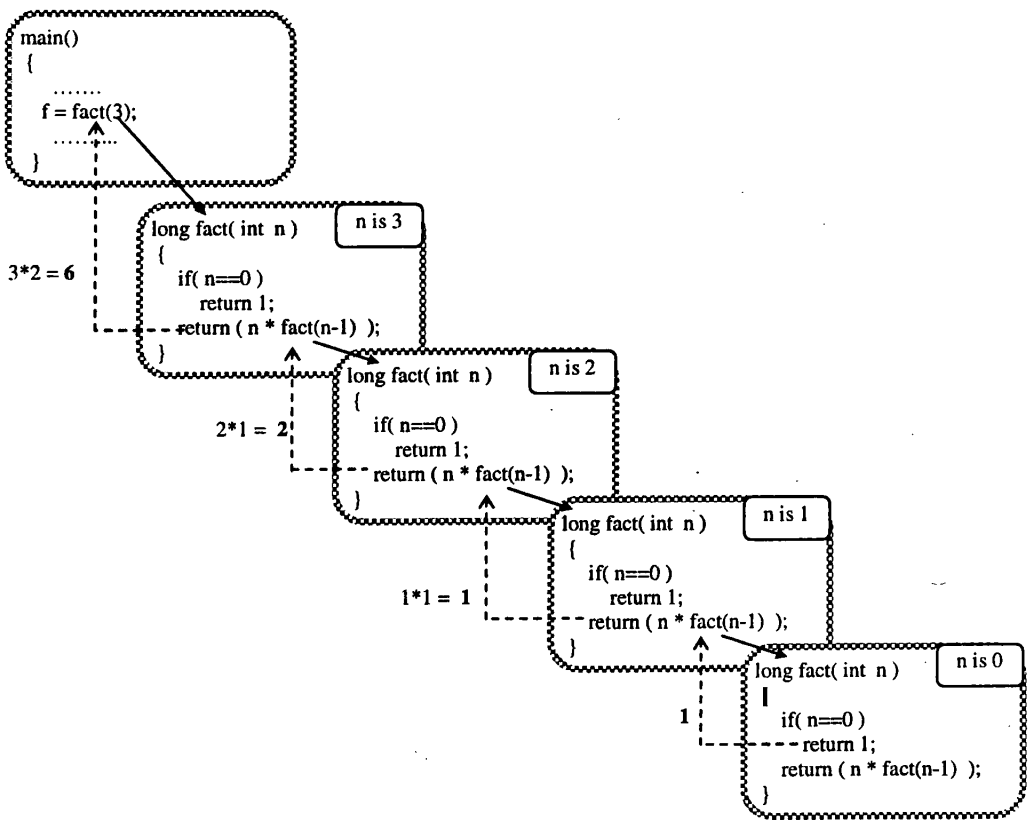


Figure 5.3 Finding Factorial of a number recursively

The function `fact()` returns 1 if the argument `n` is 0, otherwise it returns `n*fact(n-1)`. To return `n*fact(n-1)`, the value of `fact(n-1)` has to be calculated for which `fact()` has to be called again with an

argument of n-1. This process of calling fact() continues till it is called with an argument of 0. Suppose we want to find out the factorial of 3.

Initially main() calls fact(3)
Since 3>0, fact(3) calls fact(2)
Since 2>0, fact(2) calls fact(1)
Since 1>0, fact(1) calls fact(0)
} winding phase

First time when fact() is called, its argument is the actual argument given in the main() which is 3. So in the first invocation of fact() the value of n is 3. Inside this first invocation, there is a call to fact() with argument n-1, so now fact() is invoked for the second time and this time the argument is 2. Now the second invocation calls third invocation of fact() and this time argument is 1. The third invocation of fact() calls the fourth invocation with an argument of 0.

When fact() is called with n=0, the condition inside if statement becomes true, i.e. we have reached the base case, so now the recursion stops and the statement return 1 is executed. The winding phase terminates here and the unwinding phase begins and control starts returning towards the original call.

Now every invocation of fact() will return a value to the previous invocation of fact(). These values are returned in the reverse order of function calls.

Value returned by fact(0) to fact(1) = 1
Value returned by fact(1) to fact(2) = 1 * fact(0) = 1*1 = 1
Value returned by fact(2) to fact(3) = 2 * fact(1) = 2*1 = 2
Value returned by fact(3) to main() = 3 * fact(2) = 3*2 = 6
} Unwinding Phase

The function fact() is called 4 times, each function call is different from another and the argument supplied is different each time.

In the program we have taken long int as the return value of fact(), since the value of factorial for even small value of n exceeds the range of int data type.

5.4.2 Summation of numbers from 1 to n

The next example of a recursive function that we are taking is a function that computes the sum of integers from 1 to n. This is similar to the factorial problem; there we had to find the product of numbers from 1 to n, while here we have to find the sum.

We can say that the sum of the numbers from 1 to n is equal to n plus the sum of numbers 1 to n-1.

summation(n) = n + summation(n-1)

summation(4) = 4 + summation(3)
= 4 + 3 + summation (2)
= 4 + 3 + 2 + summation (1)
= 4 + 3 + 2 + 1 + summation (0)

We know that summation(0) will be equal to 0, and this can act as the base case. So the base case occurs when the function is called with value of n equal to 0 and in this case the function should return 0.

```
int summation(int n)
{
        if(n==0)
                return 0;

        return (n + summation(n-1));
}/*End of summation()*/
```

5.4.3 Displaying numbers from 1 to n

In the previous two examples we had found out the product and sum of numbers from 1 to n, now we will write a recursive function to display these numbers. This function has to only display the numbers so it will not return any value and will be of type `void`.

We have written two functions `display1()` and `display2()`, these functions are traced in figures 5.4 and 5.5. Let us see which one gives us the desired output.

```
void display1(int n)
{
        if(n==0)
           return;
        printf("%d ",n);
        display1(n-1);
}/*End of display1()*/

void display2(int n)
{
        if(n==0)
           return;
        display2(n-1);
        printf("%d ",n);
}/*End of display2()*/
```

The function `display1()` is traced in figure 5.4. Initially the `printf()` inside the first invocation will be executed and in that invocation value of n is 3, so first 3 is displayed. After this the `printf()` inside second invocation is executed and 2 is displayed, and then `printf()` inside third invocation is executed and 1 is displayed. In the fourth invocation the value of n becomes 0, we have reached the base case and the recursion is stopped. So this function displays the values from n to 1, but we wanted to display the values from 1 to n.

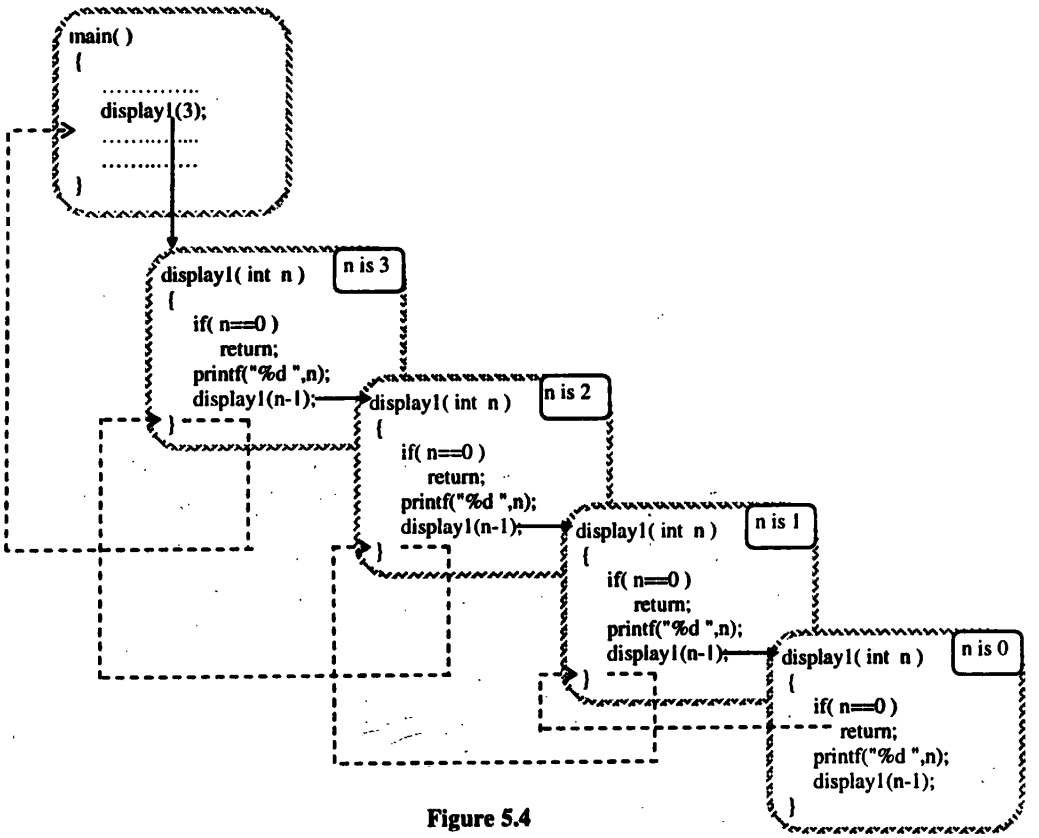


Figure 5.4

Now let us trace the function `display2()`.

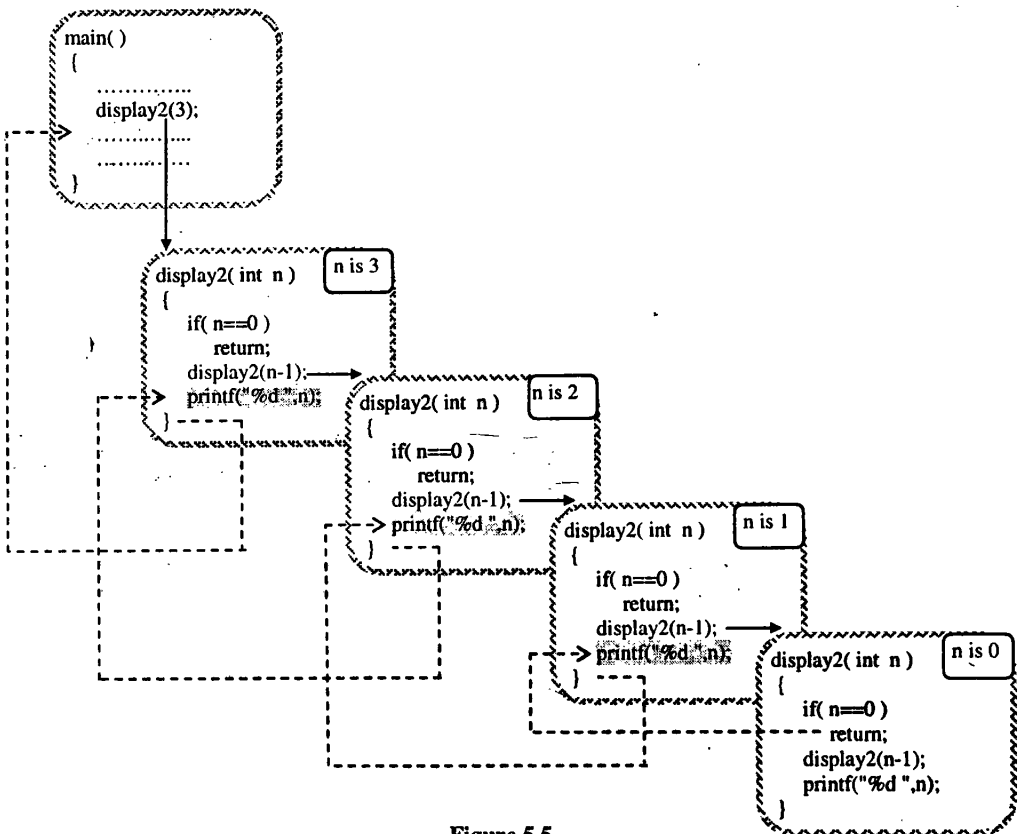


Figure 5.5

In `display2()`, the `printf()` is after the recursive call so the `printf()` functions are not executed in the winding phase, but they are executed in the unwinding phase. In unwinding phase, the calls return in reverse order, so first of all the `printf()` of third invocation is executed and in this invocation value of `n` is 1, so 1 is displayed. After this the `printf()` of second invocation is executed and in this invocation value of `n` is 2 so 2 is displayed and at last the `printf()` of first invocation is executed and 3 is displayed.

5.4.4 Display and Summation of series

Let us take a simple series of numbers form 1 to n. We want to write a recursive function that displays this series as well as finds the sum of this series. For example if the number of terms is 5, then our function should give this output.

$1 + 2 + 3 + 4 + 5 = 15$

We have already seen how to display and find the sum of numbers from 1 to n so let us combine that logic to write a function for our series.

```
/*P5.3 Program to display and find out the sum of series*/
/* Series : 1 + 2 + 3 + 4 + 5 +..... */
#include<stdio.h>
int rseries(int n);
main( )
{
        int n;
```

```
        printf("Enter number of terms : ");
        scanf("%d", &n);

        printf("\b\b = %d\n\n\n", rseries(n)); /* \b to erase last + sign*/
}/*End of main()*/
int rseries(int n)
{
        int sum;
        if (n == 0)
                return 0;
        return n + rseries(n-1);
        printf("%d + ",n);
}/*End of rseries()*/
```

Let us find out whether the function `rseries()` gives us the desired output or not. This function will return the sum of the series but will not display any term of the series. This is because in the unwinding phase when control returns to the previous invocation, the return statement
`(return n + rseries(n-1))` is executed and so the function returns without executing the `printf()` function. To make the function work correctly we can write it like this -

```
int rseries(int n)
{
        int sum;
        if(n == 0)
                return 0;
        sum = (n + rseries(n-1));
        printf("%d + ",n);
        return sum;
}/*End of rseries()*/
```

The `printf()` is before the return statement and after the recursive call, so it will always be executed in the unwinding phase.

5.4.5 Sum of digits of an integer and displaying an integer as sequence of characters

The problem of finding sum of digits of a number can be defined recursively as-

sumdigits(n) = least significant digit of n + sumdigits (n with least significant digit removed)

The sum of digits of a single digit number is the number itself, and it can act as the base case.

If we have to find sum of digits or 45329, we can proceed as-

sumdigits(45329) = 9 + sumdigits(4532)
sumdigits(4532) = 2 + sumdigits(453)
sumdigits(453) = 3 + sumdigits(45)
sumdigits(45) = 5 + sumdigits(4)
sumdigits(4) = 4

The least significant digit of the integer n can be extracted by the expression `n%10`. The recursive call has to be made with the least significant digit removed and this is done by calling the function with `(n/10)`. The base case would be when the function is called with a one digit argument.

```
/*Finds the sum of digits of an integer*/
int sumdigits(long int n)
{
        if(n/10 == 0) /* if n is a single digit number*/
                return n;
        return n%10 + sumdigits(n/10);
}/*End of sumdigits()*/
```

```
/*Displays the digits of an integer*/
void display(long int n)
{
        if(n/10 == 0)
        {
                printf("%d",n);
                return;
        }
        display(n/10);
        printf("%d",n%10);
}/*End of display()*/
/*Displays the digits of an integer in reverse order*/
void Rdisplay(long int n)
{
        if(n/10==0)
        {
                printf("%d",n);
                return;
        }
        printf("%d",n%10);
        Rdisplay(n/10);
}/*End of Rdisplay()*/
```

5.4.6 Base conversion

Now we will write a recursive function which will convert a decimal number to binary, octal and hexadecimal base. To do this conversion we have to divide the decimal number repeatedly by the base till it is reduced to 0 and print the remainders in reverse order. If the base is hexadecimal then we have to print alphabets for remainder values greater than or equal to 10. We want to print the remainders in reverse order, so we can do this work in the unwinding phase.

```
/*P5.5 Program to convert a positive decimal number to Binary, Octal or Hexadecimal*/
#include<stdio.h>
void convert(int, int);
main()
{
        int num;
        printf("Enter a positive decimal number : ");
        scanf("%d",&num);
        convert(num, 2);        printf("\n");
        convert(num, 8);        printf("\n");
        convert(num, 16);       printf("\n");
}/*End of main()*/
void convert(int num,int base)
{
        int rem = num%base;
        if(num==0)
                return;
        convert(num/base,base);

        if(rem < 10)
                printf("%d",rem);
        else
                printf("%c",rem-10+'A');
}/*End of convert()*/
```

5.4.7 Exponentiation of a float by a positive integer

The iterative definition for finding a^n is

$a^n = a * a * a * \ldots\ldots\ldots$ n times

The recursive definition can be written as-

$$a^n = \begin{cases} 1 & n=0 \quad \text{(Base case)} \\ a * a^{n-1} & n>0 \quad \text{(Recursive case)} \end{cases}$$

```
/*P5.6 Program to raise a floating point number to a positive integer*/ #include<stdio.h>
float power(float a,int n);
main()
{
        float a,p;
        int n;
        printf("Enter a and n : ");
        scanf("%f%d",&a,&n);
        p = power(a,n);
        printf("%f raised to power %d is %f\n",a,n,p);
}/*End of main()*/
float power(float a,int n)
{
        if(n == 0)
                return(1);
        else
                return(a * power(a, n-1));
}/*End of power()*/
```

5.4.8 Prime Factorization

Prime factorization of a number means factoring a number into a product of prime numbers. For example prime factors of numbers 84 and 45 are -

84 = 2 * 2 * 3 * 7
45 = 3 * 3 * 5

To find prime factors of a number n, we check its divisibility by prime numbers 2,3,5,7,... till we get a divisor d. Now d becomes a prime factor and the problem reduces to finding prime factors of n/d. The base case occurs when problem reduces to finding prime factors of 1. Let us see how we can find prime factors of 34.

84 is divisible by 2, so 2 is a PF and now problem reduces to finding PFs of 84/2 = 42
42 is divisible by 2, so 2 is a PF and now problem reduces to finding PFs of 42/2=21
21 is not divisible by 2 so we check divisibility by next prime number which is 3,
21 is divisible by 3, so 3 is PF and now problem reduces to finding PFs of 21/3=7.
7 is not divisible by 3 so we check divisibility by next prime number which is 5,
7 is not divisible by 5 so we check divisibility by next prime number which is 7,
7 is divisible by 7, so 7 is a PF and now problem reduces to finding PFs of 7/7=1

The recursive function for printing the prime factors of a number are-

```
void PFactors(int num)
{
        int i = 2;
        if(num == 1)
                return;
        while(num%i != 0)
                i++;
        printf("%d ",i);
        PFactors(num/i);
}/*End of PFactors()*/
```

For simplicity we have checked the divisibility by all numbers starting from 2(prime and non prime), but we will get only prime factors. For example 6 is non prime factor of 84 but it has already been removed as a factor of 2 and factor of 3.

5.4.9 Greatest Common Divisor

The greatest common divisor (or highest common factor) of two integers is the greatest integer that divides both of them without any remainder. It can be found out by using Euclid's remainder algorithm which states that-

$$GCD(a, b) = \begin{cases} a & b = 0 \quad \text{(Base case)} \\ GCD(b, a\%b) & \text{otherwise} \quad \text{(Recursive case)} \end{cases}$$

GCD(35, 21) (a = 35, b = 21)
= GCD(21, 14) (a = 21, b = 35 % 21 = 14)
= GCD(14, 7) (a = 14, b = 21 % 14 = 7)
= GCD(7, 0) (a = 7, b = 14 % 7 = 0)
= 7

GCD(12, 26) (a = 12, b = 26)
= GCD(26, 12) (a = 26, b = 12 % 26 = 12)
= GCD(12, 2) (a = 12, b = 26 % 12 = 2)
= GCD(2, 0) (a = 2, b = 12 % 2 = 0)
= 2

The recursive function can be written as-

```
int GCD(int a, int b)
{
      if(b==0)
         return a;
      return GCD(b, a%b);
}/*End of GCD()*/
```

Here nothing is to be done in the unwinding phase. The value returned by the last recursive call just becomes the return value of previous recursive calls, and finally it becomes the return value of the first recursive call.

5.4.10 Fibonacci Series

Fibonacci series is a sequence of numbers in which the first two numbers are 1, and after that each number is the sum of previous two numbers.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The problem of finding the n^{th} Fibonacci number can be recursively defined as-

$$fib(n) = \begin{cases} 1 & n=0 \text{ or } n=1 \quad \text{(Base case)} \\ fib(n-1) + fib(n-2) & n>1 \quad \text{(Recursive case)} \end{cases}$$

```
/*P5.9 Program to generate fibonacci series*/
int fib(int n);
main()
{
```

```
        int nterms, i;
        printf("Enter number of terms : ");
        scanf("%d", &nterms);
        for(i=0; i<nterms; i++)
                printf("%d ",fib(i));
        printf("\n");
}/*End of main()*/
int fib(int n)
{
        if(n==0  || n==1)
                return(1);
        return(fib(n-1) + fib(n-2));
}/*End of fib()*/
```

Here the function `fib()` is called two times inside its own function body. The following figure shows the recursive calls of function `fib()` when it is called with argument 5.

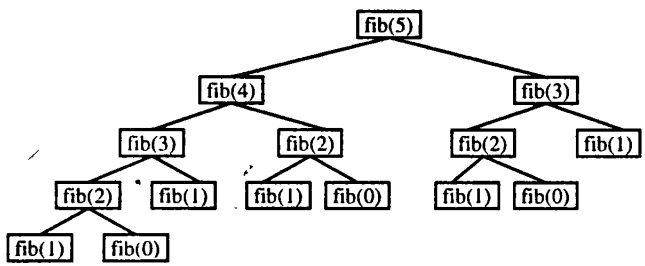


Figure 5.6

This implementation of Fibonacci is very inefficient because it performs same computations repeatedly, for example in the above example it computed the value of `fib(2)` 3 times.

5.4.11 Checking Divisibility by 9 and 11

We can easily check divisibility by any number using % operator but here we will develop functions for checking divisibility by 9 and 11 using the definitions that we have learnt in mathematics because these can help us improve our recursive thinking.
A number is divisible by 9 if and only if the sum of digits of the number is divisible by 9.

test(4968589) -> 4 + 9 + 6 + 9 + 8 + 5 + 8 + 9 = 58

test(58) -> 5 + 8 = 13

test (13) -> 1 + 3 = 4

test(4) -> not divisible by 9

test(1469358) -> 1 + 4 + 6 + 9 + 3 + 5 + 8 = 36

test(36) -> 3 + 6 = 9

test(9) -> divisible by 9

The function will be recursively called for the sum of digits of the number. The recursion will stop when the number reduces to one digit. If that digit is 9, number is divisible by 9. If that digit is less than 9 then number is not divisible by 9.

```
int divisibleBy9(long int n)
{
        int sumofDigits;
        if(n==9)
                return 1;
```

```
        if(n<9)
                return 0;
        sumofDigits = 0;
        while(n>0)
        {
                sumofDigits += n%10;
                n/=10;
        }
        divisibleBy9(sumofDigits);
}/*End of divisibleBy9()*/
```

This function returns 1 if number is divisible by 9 otherwise it returns 0.

Now let us make a function that checks divisibility by 11. A number is divisible by 11 if and only if the difference of the sums of digits at odd positions and even positions is either zero or divisible by 11.

test(91628153) -> [**28**(9+6+8+5) – **7**(1+2+1+3)] = 21
test(21) -> [2 – 1) = 1
test(1) -> Not divisible by 11

62938194 -> [**32**(6+9+8+9) – **10**(2+3+1+4)] = 22
test(22) -> [2 – 2] = 0
test(0) -> divisible by 11

The function will be recursively called for the difference of digits in odd and even positions. The recursion will stop when the number reduces to one digit. If that digit is 0, then number is divisible by 11 otherwise it is not divisible by 11.

```
int divisibleBy11(long int n)
{
        int s1=0,s2=0,diff;
        if(n==0)
                return 1;
        if(n<10)
                return 0;
        while(n>0)
        {
                s1 += n%10;
                n/=10;
                s2 += n%10;
                n/=10;
        }
        diff = s1>s2 ? (s1-s2) : (s2-s1);
        divisibleBy11(diff);
}/*End of divisibleBy11()*/
```

This function returns 1 if number is divisible by 11 otherwise it returns 0.

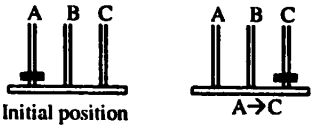
5.4.12 Tower of Hanoi

The problem of Tower of Hanoi is to move disks from one pillar to another using a temporary pillar. Suppose we have a source pillar A which has finite number of disks, and these disks are placed on it in a decreasing order i.e. largest disk is at the bottom and the smallest disk is at the top. Now we want to place all these disks on destination pillar C in the same order. We can use a temporary pillar B to place the disks temporarily whenever required. The conditions for this game are-

1. We can move only one disk from one pillar to another at a time.
2. Larger disk cannot be placed on smaller disk.

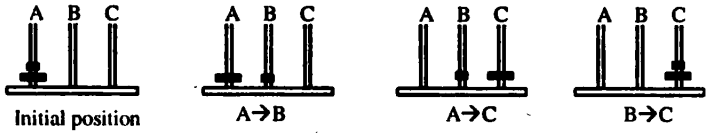
Suppose the number of disks on pillar A is n. First we will solve the problem for n=1, n=2, n=3 and then we will develop a general procedure for the solution. Here A is the source pillar, C is the destination pillar and B is the temporary pillar.

For n=1



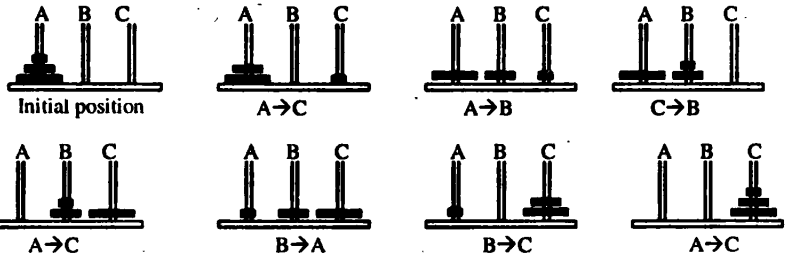
Move the disk from pillar A to pillar C. (A→C)

For n=2



(i) Move disk 1 from pillar A to B (A→B)
(ii) Move disk 2 from A to C (A→C)
(iii) Move disk 1 from pillar B to C (B→C)

For n=3



(i) Move disk 1 from pillar A to C (A→C)
(ii) Move disk 2 from pillar A to B (A→B)
(iii) Move disk 1 from pillar C to B (C→B)
(iv) Move disk 3 from pillar A to C (A→C)
(v) Move disk 1 from pillar B to A (B→A)
(vi) Move disk 2 from pillar B to C (B→C)
(vii) Move disk 1 from pillar A to C (A→C)

These were the solutions for n=1, n=2, n=3. From these solutions we can observe that first we move n-1 disks from source pillar (A) to temporary pillar (B) and then move the largest(n^{th}) disk to the destination pillar(C). So the general solution for n disks can be written as-

1. Move upper n-1 disks from A to B using C as the temporary pillar.
2. Move n^{th} disk from A to C.
3. Move n-1 disks from B to C using A as the temporary pillar.

The base case would be when we have to move only one disk, in that case we can simply move the disk from source to destination pillar and return. The recursion tree for n=5 is given below.

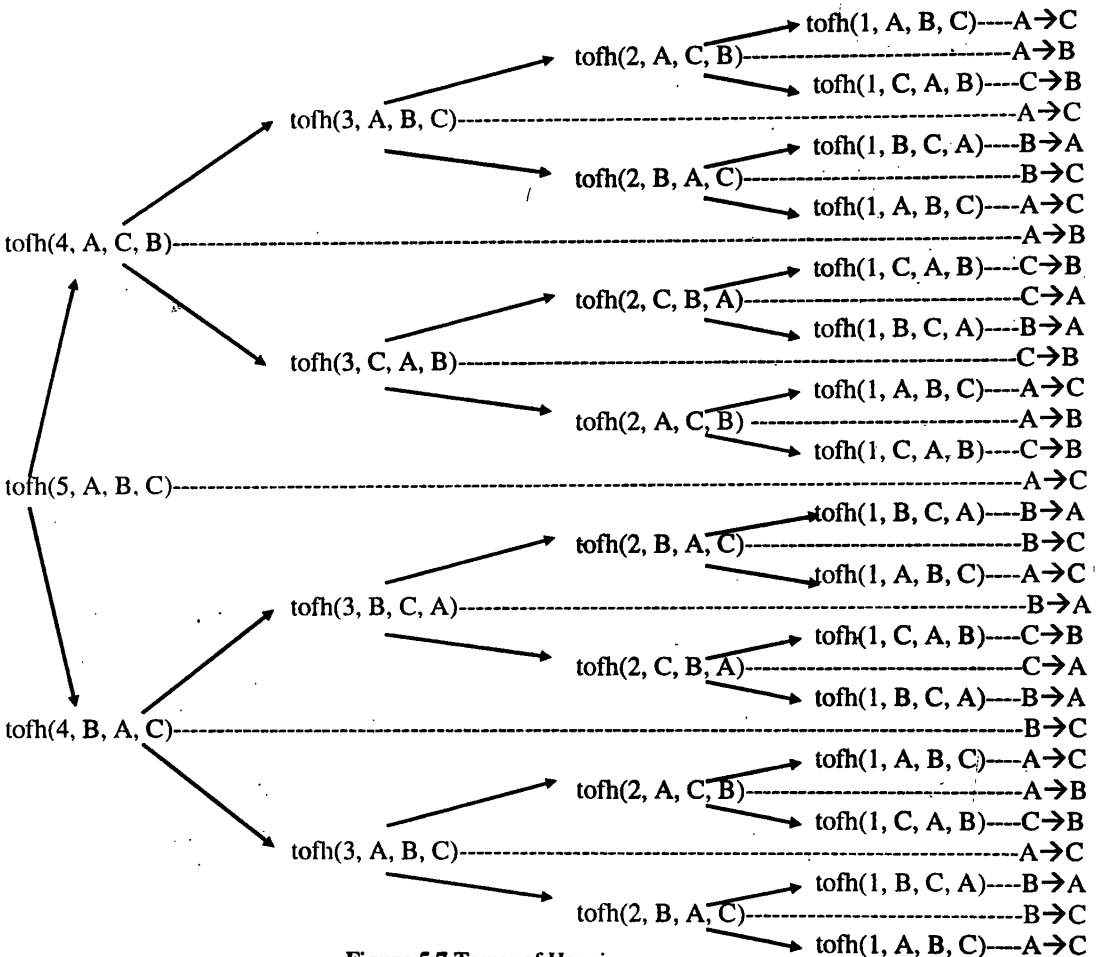


Figure 5.7 Tower of Hanoi

The recursive function for solving tower of Hanoi can be defined as-

$$\text{tofh}(n, \text{source}, \text{temp}, \text{dest}) = \begin{cases} \text{Move disk 1 from source to dest} & n=1 \\ \text{tofh}(n-1, \text{source}, \text{dest}, \text{temp}) \\ \text{Move } n^{th} \text{ disk from source to dest} & n>1 \\ \text{tofh}(n-1, \text{temp}, \text{source}, \text{dest}) \end{cases}$$

```
/*P5.11 Program to solve Tower of Hanoi problem using recursion*/
#include<stdio.h>
void tofh(int ndisk,char source,char temp,char dest);
main()
{
        char source='A',temp='B',dest='C';
        int ndisk;
        printf("Enter the number of disks : ");
        scanf("%d", &ndisk);
        printf("Sequence is :\n");
        tofh(ndisk,source,temp,dest);

}
void tofh(int ndisk,char source,char temp,char dest)
```

```
{
        if(ndisk==1)
        {
                printf("Move Disk %d from %c-->%c\n",ndisk,source,dest);
                return;
        }
        tofh(ndisk-1,source,dest,temp);
        printf("Move Disk %d from %c-->%c\n",ndisk,source,dest);
        tofh(ndisk-1,temp,source,dest);
}/*End of tofH()*/
```

5.5 Recursive Data structures

A recursive definition of a data structure defines the data structure in terms of itself. Some data structures like strings, linked lists, trees can be defined recursively. In these types of data structures we can take advantage of the recursive structure, and so the operations on these data structures can be naturally implemented using recursive functions. This type of recursion is called structural recursion.

A string can be defined recursively as-
1. A string may be an empty string.
2. A string may be a character followed by a smaller string(one character less).

The string "leaf" is character 'l' followed by string "eaf", similarly string "eaf" is character 'e' followed by string "af", string "af" is character 'a' followed by string "f", string "f" is character 'f' followed by empty string. So while defining recursive operations on strings, case of empty string will serve as the base case for terminating recursion.

Similarly we can define a linked list recursively as-
(i) A linked list may be an empty linked list.
(ii) A linked list may be a node followed by a smaller linked list(one node less).

If we have a linked list containing nodes N1, N2, N3, N4 then we can say that

linkedlist(N1-> N2-> N3-> N4) is node N1 followed by linked list(N2-> N3-> N4),
linkedlist(N2-> N3-> N4) is node N2 followed by linked list(N3-> N4),
linkedlist(N3-> N4) is node N3 followed by linked list(N4),
linkedlist(N4) is node N4 followed by an empty linked list.

While implementing operations on linked lists we can take the case of empty linked list as the base case for stopping recursion.

In this chapter we will study some recursive procedures to operate on strings and linked lists. In the next chapter we will study about another data structure called tree which can also be defined recursively. Some operations on trees are best defined recursively, so before going to that it is better if you understand recursion in linked lists.

5.5.1 Strings and recursion

We have seen the recursive definition of strings and the base case. Now we will write some recursive functions for operations on strings. The first one is to find the length of a string.
The length of empty string is 0 and this will be our terminating condition. In general case, the function is recursively called for a smaller string, which does not contain the first character of the string.

```
int length(char *str)
{
        if(*str == '\0')
                return 0;
        return (1 + length(str+1));
}/*End of length()*/
```

We can print a string by printing the first character of the string followed by printing of the smaller string, if the string is empty there is nothing to be printed so we will do nothing and return(base case).

```
void display(char *str)
{
        if(*str == '\0')
              return;
        putchar(*str);
        display(str+1);
}/*End of display()*/
```

By changing the order of printing statement and recursive call we can get the function for displaying string in reverse order.

```
void Rdisplay(char *str)
{
        if(*str == '\0')
              return;
        Rdisplay(str+1);
        putchar(*str);
}/*End of Rdisplay()*/
```

Note that for displaying string in standard order we have placed the printing statement before the recursive call while during the display of numbers from 1 to n (Section 5.4.3), we had placed the printing statement after the recursive call. We hope that you can understand the reason for this and if not then trace and find out.

5.5.2 Linked lists and recursion

The first recursive function for linked list that we will make is a function to find out the length of the linked list. Length of a linked list is 1 plus the length of smaller list(list without the first node) and length of empty list is zero(base case).

```
int length(struct node *ptr)
{
        if(ptr==NULL)
              return 0;
        return 1 + length(ptr->link);
}/*End of length()*/
```

Similarly we can make a function to find out the sum of the elements of linked list.

```
int sum(struct node *ptr)
{
        if(ptr==NULL)
              return 0;
        return ptr->info + sum(ptr->link);
}/*End of sum()*/
```

Next we will make a function for printing the elements of a linked list. We can print a list by printing the first element of list followed by printing of the smaller list, if the list is empty there is nothing to be printed so we will do nothing and return.

```
void display(struct node *ptr)
{
        if(ptr==NULL)
              return;
        printf("%d ",ptr->info);
        display(ptr->link);
}/*End of display()*/
```

We are just walking down the list till we reach NULL, and printing the info part in the winding phase. This function will be invoked as –

```
display(start);
```

Next we will make a function to print the list in reverse order, i.e. it will print all the elements starting from the last element.

```
void Rdisplay(struct node *ptr)
{
        if(ptr==NULL)
                return;
        Rdisplay(ptr->link);
        printf("%d ",ptr->info);
}/*End of Rdisplay()*/
```

The next function searches for an element in the list and if it is present it returns 1, otherwise it returns 0.

```
int search(struct node *ptr, int item)
{
        if(ptr==NULL)
                return 0;
        if(ptr->info==item)
                return 1;
        return search(ptr->link,item);
}/*End of search()*/
```

The next function inserts a node at the end of the linked list.

```
struct node *insertLast(struct node *ptr,int item)
{
        struct node *temp;
        if(ptr==NULL)
        {
                temp = malloc(sizeof(struct node));
                temp->info = item;
                temp->link = NULL;
                return temp;
        }
        ptr->link = insertLast(ptr->link, item);
      return ptr;
}/*End of insertLast()*/
```

This function will be invoked as `start = insertLast(start);`

The next function deletes the last node from the linked list.

```
struct node *delLast(struct node *ptr)
{
        if(ptr->link==NULL)
        {
                free(ptr);
                return NULL;
        }
        ptr->link = delLast(ptr->link);
        return ptr;
}/*End of delLast()*/
```

This function will be invoked as `start = delLast(start);`

The next function reverses the linked list.

```
struct node *reverse(struct node *ptr)
{
        struct node *temp;

        if(ptr->link==NULL)
                return ptr;

        temp=reverse(ptr->link);
        ptr->link->link=ptr;
        ptr->link=NULL;
        return temp;
```

```
}/*End of reverse()*/
```

This function will be invoked as `start = reverse(start);`

5.6 Implementation of Recursion

We have seen that recursive calls execute just like normal function calls, so there is no special technique for implementing them. All function calls whether recursive or non recursive are implemented through run time stack and in the previous chapter we had seen how it is done. Here we will take the example of function `fact()` called by main with argument 3, and see the changes in the run time stack as the function `fact()` is evaluated.

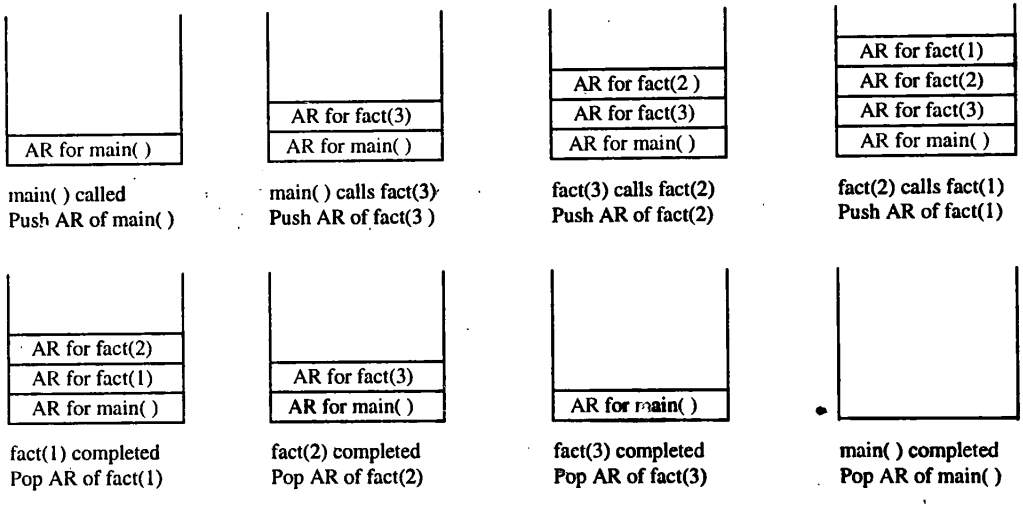


Figure 5.8

In the winding phase, the stack grows as new activation records are created and pushed for each invocation of the function. In the unwinding phase, the activation records are popped from the stack in LIFO manner till the original call returns.

5.7 Recursion vs. Iteration

All repetitive problems can be implemented either recursively or iteratively. First we will compare the workings of both approaches.

In loops, the same block of code is executed repeatedly, and repetition occurs when the block of code is finished or a continue statement is encountered. In recursion, the same block of code is repeatedly executed and repetition occurs when the function calls itself.

In loops, the variables inside the loop are modified using some update statement. In recursion, the new values are passed as parameters to the next recursive call.

For the loop to terminate, there is a terminating condition and the loop progresses in such a way that this condition is eventually hit. If this does not happen then we get an infinite loop. For the recursion to terminate, there is a terminating condition(base case) when function stops calling itself, and the recursion should proceed in such a way that we finally hit the base case. If this does not happen then we will have infinite recursion, and the function will keep on calling itself till the stack is exhausted and we get stack overflow error.

Recursion involves pushing and popping activation records of all the currently active recursive calls on the stack. So the recursive version of a problem is usually slower because of the time spent in pushing and popping these activation records. It is also expensive in terms of memory as it uses space in run time stack to store these activation records. If the recursion is too deep, then the stack may overflow and the program will crash. On the other hand the iterative versions do not have to pay for this function call overhead and so are faster and require less space.

Recursive solutions involve more execution overhead than their iterative counterparts, but their main advantage is that they simplify the code and make it more compact and elegant. Recursive algorithms are easier to understand because the code is shorter and clearer.

Recursion should be used when the underlying problem is recursive in nature or when the data structure on which we are operating is recursively defined like trees. Iteration should be used when the problem is not inherently recursive, or the stack space is limited.

For some problems which are very complex, iterative algorithms are harder to implement and it is easier to solve them recursively. In these cases, recursion offers a better way of writing our code which is both logical and easier to understand and maintain. So sometimes it may be worth sacrificing efficiency for code readability.

Recursion can be removed by maintaining our own stack or by using iterative version.

5.8 Tail recursion

Before studying about tail recursive functions, let us see what tail recursive calls are. A recursive call is tail recursive if it is the last statement to be executed inside the function.

```
void display1(int n)
{
        if(n==0)
           return;
        printf("%d ",n);
        display1(n-1); /*Tail recursive Call*/
}/*End of display1()*/
void display2(int n)
{
        if(n==0)
           return;
        display2(n-1); /*Not a Tail recursive Call*/
        printf("%d ",n);
}/*End of display2()*/
```

In non void functions, if the recursive call appears in return statement and that call is not part of an expression then the call is tail recursive.

```
int GCD(int a,int b)
{
        if(b==0)
           return a;
        return GCD(b,a%b);  /*Tail recursive call*/
}/*End of GCD()*/
long fact(int n)
{
        if(n==0)
                return(1);
        return(n * fact(n-1)); /*Not a tail recursive call*/
}/*End of fact()*/
```

Here the call `fact(n-1)` appears in the return statement but it is not a tail recursive call because the call is part of an expression. Now let us see some functions which have more than one recursive calls.

```
void tofh(int ndisk,char source,char temp,char dest)
{
        if(ndisk==1)
        {
                printf("Move Disk %d from %c-->%c\n",ndisk,source,dest);
                return;
        }
        tofh( ndisk-1,source,dest,temp ); /*Not tail recursive call*/
        printf("Move Disk %d from %c-->%c\n",ndisk,source,dest);
        tofh(ndisk-1,temp,source,dest ); /*Tail recursive call*/
}/*End of tofh()*/
```

Here the first recursive call is not a tail recursive call while the second one is a tail recursive call. In the next function we have two recursive calls and both are tail recursive.

```
int binary_search(int arr[],int item, int low, int up)
{
        int mid;
        if(up<low)
                return -1;
        mid = (low+up)/2;
        if(item>arr[mid])
                return binary_search(arr,item,mid+1,up);/*tail recursive call*/
        else if(item<arr[mid])
                return binary_search(arr,item,low,mid-1);/*tail recursive call*/
        else
                return mid;
}/*End of binary_search()*/
```

Here only one recursive call will be executed in each invocation of the function, and that recursive call will be the last one to be executed inside the function. So both the calls in this function are tail recursive.

The two recursive calls in `fibonacci()` function(P5.9) are not tail recursive because these calls are part of an expression, and after returning from the call, the return value has to be added to the return value of other recursive call.

A function is tail recursive if all the recursive calls in it are tail recursive. In the examples given earlier the tail recursive functions are - `display1()`, `GCD()`, `binary_search()`. The functions `display2()`, `tofh()`, `fact()` are not tail recursive functions.

Tail recursive functions can easily be written using loops, because as in a loop there is nothing to be done after an iteration of the loop finishes, in tail recursive functions there is nothing to be done after the current recursive call finishes execution. Some compilers automatically convert tail recursion to iteration for improving the performance.

In tail recursive functions, the last work that a function does is a recursive call, so no operation is left pending after the recursive call returns. In non void tail recursive functions(like GCD) the value returned by the last recursive call is the value of the function. Hence in tail recursive functions, there is nothing to be done in the unwinding phase.

Since there is nothing to be done in the unwinding phase, we can jump directly from the last recursive call to the place where recursive function was first called. So there is no need to store the return address of previous recursive calls and values of their local variables, parameters, return values etc. In other words there is no need of pushing new AR for all recursive calls.

Some modern compilers can detect tail recursion and perform tail recursion optimization. They do not push a new activation record when a recursive call occurs; rather they overwrite the previous activation record by current activation record, while retaining the original return address. So we have only one activation record in the stack at a time, and this is for the currently executing recursive call. This improves the performance by reducing the time and memory requirement. Now it doesn't matter how deep the recursion is, the space required will be always be constant.

Since tail recursion can be efficiently implemented by compliers, we should try to make our recursive functions tail recursive whenever possible.

A recursive function can be written as a tail recursive function using an auxiliary parameter. The result is accumulated in this parameter and this is done in such a way that there is no pending operation left after the recursive call. For example we can rewrite the factorial function that we have written earlier as a tail recursive function.

```
long TRfact(int n, int result)
{
        if(n==0)
                return result;
        return TRfact(n-1, n*result);
}/*End of TRfact()*/
```

This function should be called as `TRfact(n,1)`. We can make a helper function to initialize the value of result to 1. The use of this helper function hides the auxiliary parameter.

```
TailRecursiveFact(int n)
{
        return TRfact(n,1);
}/*End of TailRecursiveFact()*/
```

Functions which are not tail recursive are called augmentive recursive functions and these types of functions have to finish the pending work after the recursive call finishes.

5.9 Indirect and direct Recursion

If a function `f1()` calls `f2()` and the function `f2()` in turn calls `f1()`, then this is indirect recursion, because the function `f1()` is calling itself indirectly.

```
f1()
{
        .....
        f2();
        .....
}
f2()
{
        .....
        f1();
        .....
}
```

The chain of functions in indirect recursion may involve any number of functions, for example `f1()` calls `f2()`, `f2()` calls `f3()`, `f3()` calls `f4()`, `f4()` calls `f1()`. If a function calls itself directly i.e. `f1()` is called inside its own function body, then that recursion is direct recursion. All the examples that we have seen in this chapter use direct recursion. Indirect recursion is complex and is rarely used.

Exercise

Find the output of programs from 1 to 16.

1.
```
main()
{
      printf("%d  %d\n",func1(3,8),func2(3,8));
}
func1(int a,int b)
{
      if(a>b)
            return 0;
      return b + func1(a,b-1);
}
func2(int a,int b)
{
      if(a>b)
            return 0;
      return a + func2(a+1,b);
}
```

2.
```
main()
{
      printf("%d \n",func(3,8));
}
int func(int a,int b)
{
```

```
    if(a>b)
            return 1000;
    return a + func(a+1,b);
}
```

3.
```
main()
{
    printf("%d\n",func(6));
    printf("%d\n",func1(6));
}
int func(int a)
{
    if(a==10)
            return a;
    return a + func(a+1);
}
int func1(int a)
{
    if(a==0)
            return a;
    return a + func1(a+1);
}
```

4.
```
main()
{
    printf("%d\n",func(4,8));
    printf("%d\n",func(3,8));
}
int func(int a,int b)
{
    if(a==b)
            return a;
    return a + b + func(a+1,b-1);
}
```

5.
```
main()
{
    func1(10,18);
    printf("\n");
    func2(10,18);
}
void func1(int a,int b)
{
    if(a>b)
            return;
    printf("%d  ",b);
    func1(a,b-1);
}
void func2(int a,int b)
{
    if(a>b)
            return;
    func2(a,b-1);
    printf("%d  ",b);
}
```

6.
```
main()
{
    func1(10,18);
    printf("\n");
    func2(10,18);
}
```

```
void func1(int a,int b)
{
      if(a>b)
            return;
      printf("%d  ",a);
      func1(a+1,b);
}
void func2(int a,int b)
{
      if(a>b)
            return;
      func2(a+1,b);
      printf("%d  ",a);
}
```

7.
```
main()
{
      printf("%d\n",func(3,8));
      printf("%d\n",func(3,0));
      printf("%d\n",func(0,3));
}
int func(int a,int b)
{
      if(b==0)
            return 0;
      if(b==1)
            return a;
      return a + func(a,b-1);
}
```

8.
```
main()
{
      printf("%d\n",count(17243));
}
int count(int n)
{
      if(n==0)
            return 0;
      else
            return 1 + count(n/10);
}
```

9.
```
main()
{
      printf("%d\n",func(14837));
}
int func(int n)
{
      return (n)? n%10 + func(n/10) : 0;
}
```

10.
```
main()
{
      printf("%d\n",count(123212,2));
}
int count(long int n,int d)
{
      if(n==0)
            return 0;
      else if(n%10 == d)
            return 1 + count(n/10,d);
      else
            return count(n/10,d);
```

```
}
```

11.
```
main()
{
    int arr[10]={1,2,3,4,8,10};
    printf("%d\n",func(arr,6));
}
int func(int arr[],int size)
{
    if(size==0)
        return 0;
    else if(arr[size-1]%2==0)
        return 1 + func(arr,size-1);
    else
        return func(arr, size-1);
}
```

12.
```
main()
{
    int arr[10]={1,2,3,4,8,10};
    printf("%d\n",func(arr,6));
}
int func(int arr[],int size)
{
    if(size==0)
        return 0;
    return arr[size-1] + func(arr,size-1);
}
```

13.
```
main()
{
    char str[100],a;
    printf("Enter a string :");
    gets(str);
    printf("Enter a character :");
    scanf("%c",&a);
    printf("%d\n",f(str,a));
}
int f(char *s,char a)
{
    if(*s=='\0')
        return 0;
    if(*s==a)
        return 1 + f(s+1,a);
    return f(s+1,a);
}
```

14.
```
main()
{
    func1(4);
    func2(4);
}
void func1(int n)
{
    int i;
    if(n==0)
        return;
    for(i=1; i<= n; i++)
        printf("*");
    printf("\n");
    func1(n - 1);
}
void func2(int n)
```

```
{
        int i;
        if(n==0)
                return;
        func2(n-1);
        for (i=1; i<= n; i++)
                printf("*");
        printf("\n");
}
```

15.
```
main()
{
        int arr[10]={2,3,1,4,6,34};
        printf("%d\n",func(arr,6));
}
int func(int arr[],int size)
{
        int m;
        if(size==1)
                return arr[0];
        m = func(arr, size-1);
        if(arr[size-1] < m )
                return arr[size-1];
        else
                return m;
}
```

16.
```
main()
{
        int arr[10]={3,4,2,11,8,10};
        printf("%d\n",func(arr,0,5) );

}
int func(int arr[],int low,int high)
{
        int mid, left, right;
        if(low==high)
                return arr[low];
        mid = (low+high)/2;
        left = func(arr,low,mid);
        right = func(arr,mid+1,high);
        if(left < right)
                return left;
        else
                return right;
}
```

17. Write a recursive function to input and add n numbers.

18. Write a recursive function to enter a line of text and display it in reverse order, without storing the text in an array.

19. Write a recursive function to count all the prime numbers between numbers a and b(both inclusive).

20. A positive proper divisor of n is a positive divisor of n, which is different from n. For example 1,3,5,9,15 are positive proper divisors 45, but 45 is not a proper divisor of 45.Write a recursive function that displays all the proper divisors of a number and returns their sum.

21. A number is perfect if the sum of all its positive proper divisors is equal to the number, for example 28 is a perfect number since 28 = 1 + 2 + 4 + 7 + 14. Write a recursive function that finds whether a number is perfect or not.

22. Write a recursive function to find the sum of all even numbers in an array.

23. Write a recursive function that finds the sum of all elements of an array by repeatedly partitioning it into two almost equal parts.

24. Write a function to reverse the elements of an array.
25. Write a recursive function to find whether the elements of an array are in strict ascending order or not.
26. Write a recursive function that displays a positive integer in words, for example if the integer is 2134 then it is displayed as - two one three four.
27. Write a recursive function that reverses an integer. For example if the input is 43287 then the function should return the integer 78234.
28. Write a recursive function to find remainder when a positive integer a is divided by positive integer b.
29. Write a recursive function to find quotient when a positive integer a is divided by positive integer b.
30. The computation of a^n can be made efficient if we apply the following procedure instead of multiplying n times -

$a^8 = (a^2)^4$
$a^4 = (a^2)^2$
$a^2 = (a^2)^1$
$a^1 = a * (a^2)^0$

$a^{11} = a * (a^2)^5$
$a^5 = a * (a^2)^2$
$a^2 = (a^2)^1$
$a^1 = a * (a^2)^0$

$a^{19} = a * (a^2)^9$
$a^9 = a * (a^2)^4$
$a^4 = (a^2)^2$
$a^2 = (a^2)^1$
$a^1 = a * (a^2)^0$

$a^{20} = (a^2)^{10}$
$a^{10} = (a^2)^5$
$a^5 = a * (a^2)^2$
$a^2 = (a^2)^1$
$a^1 = a * (a^2)^0$

Write a recursive function to compute a^n using this procedure.
31. Write a recursive function to multiply two numbers by Russian peasant method. Russian peasant method multiplies any two positive numbers using multiplication by 2, division by 2 and addition. Here the first number is divided by 2(integer division), and the second is multiplied by 2 repeatedly until the first number reduces to 1. Suppose we have to multiply 19 by 25, we write the result of division and multiplication by 2, in the two columns like this-

19	**25**	Add
9	**50**	Add
4	100	
2	200	
1	**400**	Add
	475	

Now to get the product we'll add those values of the right hand column, for which the corresponding left column values are odd. So 25, 50, 400 will be added to get 475, which is the product of 19 and 25.

32. Write recursive functions to find values of $\lfloor \log_2 N \rfloor$ and $\lfloor \log_b N \rfloor$.
33. Write a recursive function to find the Binomial coefficient C(n,k) which is defined as-

C(n,0)=1
C(n,n)=1
C(n,k) = C(n-1,k-1) + C(n-1,k)

34. Write a recursive function to compute Ackermann's function A(m,n) which is defined as-

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0, n=0 \\ A(m-1, A(m,n-1)) & \text{otherwise} \end{cases}$$

35. Write a recursive function to count the number of vowels in a string.
36. Write a recursive function to replace each occurrence of a character by another character in a string.
37. Write a recursive function to reverse a string.
38. Write a recursive function to return the index of first occurrence of a character in a string.
39. Write a recursive function to return the index of last occurrence of a character in a string.
40. Write a recursive function to find whether a string is palindrome or not. A palindrome is a string that is read the same way forward and backward for example "radar", "hannah", "madam".

41. In the program of previous problem, make changes so that spaces, punctuations marks, uppercase and lowercase differences are ignored. The strings "A man, a plan, a canal – Panama!", "Live Evil" should be recognized as palindromes.

42. Write a function to convert a positive integer to string.

43. Write a function to convert a string of numbers to an integer.

44. Write a function to print all possible permutations of a string. For example if the string is "abc" then all possible permutations are abc, acb, bac, bca, cba, cab.

45. Write a function to print these pyramids of numbers.

```
1          1 2 3 4        4 3 2 1
1 2        1 2 3          3 2 1
1 2 3      1 2            2 1
1 2 3 4    1              1
```

46. A triangular number is the number of dots required to fill an equilateral triangle. The first 4 triangular numbers are 1, 3, 6, 10.

```
*        *       *         *
        **      **        **
                ***       ***
                          ****
```

Write a recursive function that returns n^{th} triangular number.

47. Write a recursive function to check if two linked lists are identical. Two lists are identical if they have same number of elements and the corresponding elements in both lists are same.

48. Write a recursive function to create a copy of a single linked list.

49. Write a recursive function to print alternate nodes of a single linked list.

50. Write a recursive function to delete a node from a single linked list.

51. Write a recursive function to insert a node in a sorted single linked list.

Trees

The data structures that we have studied till now like linked list, stack, and queue are all linear data structures. Linked list is a good data structure from memory point of view, but the main disadvantage with linked list is that it is a linear data structure. Every node has information of next node only. So the searching in linked list is sequential which is very slow and of O(n). For searching any element in list, we have to visit all elements of the list that come before that element. If we have a linked list of 100000 elements and the element to be searched is not present or is present at the end of list then we have to visit all the 100000 elements. Now we will study a non linear data structure in which data is organized in a hierarchical manner. A tree structure represents hierarchical relationship among its elements. It is very useful for information retrieval and searching in it is very fast. Before going to the definition of trees, let us become familiar with the common terms used.

6.1 Terminology

We will use the tree given below in figure 6.1 to describe the terms associated with trees.

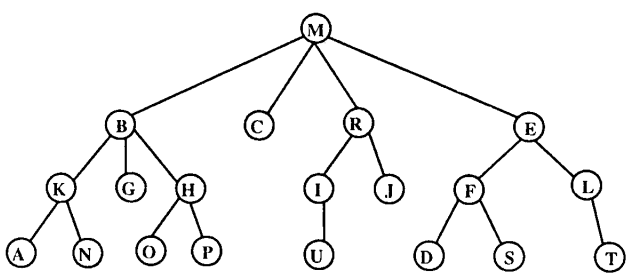


Figure 6.1 Tree

Node - Each element of tree is called a node. In the figure each node is represented by a circle.

Edges - The lines connecting the nodes are called edges or branches.

Parent Node - The immediate predecessor of a node is called parent node or father node. For example, node M is parent of nodes B, C, R, E and node I is parent of node U. Grandparent is the parent of parent, for example node E is grandparent of node D, node M is grandparent of node H.

Child Node - All the immediate successors of a node are its child nodes. For example node C is child of node M, node D is child of node F, node N is child of node K. Grandchild is the child of child node, for example node U is grandchild of node R, node L is grandchild of node M.

Root Node - This is a specially designated node that does not have any parent. In the example tree, node M is the root node.

Leaf node - A node that does not have any child is called leaf node or terminal node. All other nodes are called non leaf nodes or non terminal nodes. The nodes A, N, G, O, P, C, U, J, D, S, T are leaf nodes.

Level - Level of any node is defined as the distance of that node from the root. Root node is at a distance 0 from itself so it is at level 0. Level number of any other node is 1 more than level number of its parent node. Node M is at level 0, nodes B, C, R, E are at level 1, nodes K, G, H, I, J, F, L are at level 2, nodes A, N, O, P, U, D, S, T are at level 3.

Height – The total number of levels in a tree is the height of the tree. So height is equal to one more than the largest level number of tree. It is also sometimes known as depth of the tree. The largest level number in the example tree is 3 so its height is 3+1 = 4
(Some texts define height of tree equal to the largest level and some take root at level 1. In this book we'll use the level and height as defined here.)

Siblings - Two or more nodes which have same parent are called siblings or brothers. B, C, R, E are siblings since their parent is M. All siblings are at the same level but it is not necessary that nodes at the same level are siblings. For example K and I are at same level but they are not siblings as their parents are different.

Path - Path of a node is defined as the sequence of nodes N_1, N_2, ... N_m such that each node N_i is parent of N_{i+1} for $1 < i < m$. In a tree there is only one path between two nodes. The path length is defined as the number of edges on the path (m-1).

Ancestor and Descendent - Any node N_a is said to be the ancestor of node N_m, if node N_a lies in the unique path from root node to the node N_m. For example node E is an ancestor of node S. If node N_a is ancestor of node N_m, then node N_m is said to be descendent of node N_a.

Subtree - A tree may be divided into subtrees which can further be divided into subtrees. The first node of the subtree is called the root of the subtree. For example, the tree in figure 6.1 may be divided into 4 subtrees - subtree BKGHANOP rooted at B, subtree C rooted at C, subtree RIJU rooted at R, subtree EFLDST rooted at E. The subtree BKGHANOP can be further divided into three subtrees – subtree KAN rooted at K, subtree G rooted at G, subtree HOP rooted at H. We can see that a subtree rooted at any node X consists of all the descendents of X. The root of the subtree is used to name the subtree so instead of saying subtree BKGHANOP we generally say subtree B.

Degree - The number of subtrees or children of a node is called its degree. For example degree of node M is 4, of B is 3, of F is 2, of I is 1, and of S is 0. The degree of a tree is the maximum degree of the nodes of the tree. The degree of the example tree is 4.

Forest - A forest is a set of n disjoint trees where $n \geq 0$. If the root of a tree is removed we get a forest consisting of its subtrees. If in the example tree we remove the root M, then we get a forest with four trees.

6.2 Definition of Tree

Tree can be defined recursively as-

A tree is a finite set of nodes such that :
(i) There is a distinguished node called root and
(ii) The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, T_2, ..., T_n where each of these sets is a tree. The sets T_1, T_2, ..., T_n are the subtrees of the root.

6.3 Binary Tree

In a binary tree no node can have more than two children i.e. a node can have 0, 1 or 2 children. Each child is designated as either left child or right child. The terminology given for general tree applies to binary trees also. Like tree, a binary tree can be defined recursively as -

A binary tree is a finite set of nodes that is
(i) either empty or
(ii) consists of a distinguished node called root and remaining nodes are partitioned into two disjoint sets T_1 and T_2 and both of them are binary trees. T_1 is called left subtree and T_2 is called the right subtree.

The following five trees are examples of binary trees-

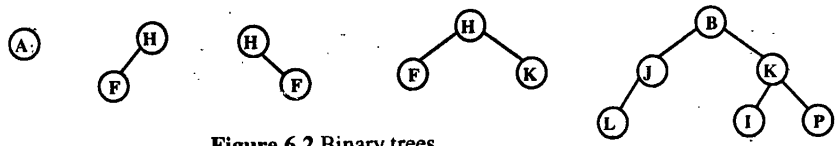


Figure 6.2 Binary trees

The first binary tree has only one node A which is the root node and it has empty left and right subtrees. The second and third binary trees have 2 nodes each but they are different because in second binary tree root has a left child while in third tree root has a right child.

Two binary trees are called similar if they have a similar structure. If the structure as well as the contents of the corresponding nodes is same then the binary trees are said to be copies.

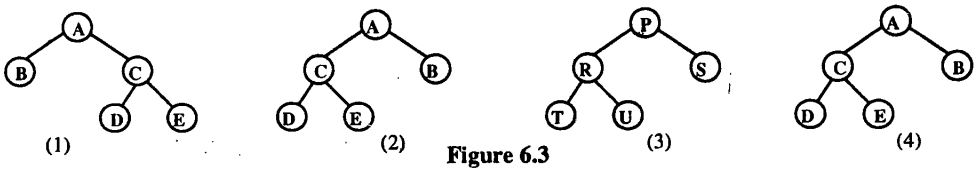


Figure 6.3

In figure 6.3, the trees (2), (3) and (4) are similar as they have same shape while trees (2) and (4) are copies as they have same shape and data. Tree (1) has different shape so it is not similar to other trees.

In binary trees, we define left and right descendent. Any node N_i is left descendent of node N_j, if N_i belongs to the left subtree of N_j, similarly any node N_i is right descendent of node N_j, if N_i belongs to the right subtree of N_j.

Now let us look at some properties of binary trees.

Property 1 - The maximum number of nodes on any level i is 2^i where $i \geq 0$

Proof : This property can be proved by induction on i. The root node is the only node on level 0. So the maximum number of nodes on level i=0 is 1 which is equal to 2^0. So the property is true for i=0. Suppose the property is true for any level k where $k \geq 0$, i.e. at level k there are at most 2^k nodes. Each node in binary tree can have at most 2 children, so the maximum number of nodes on level k+1 will be twice the maximum number of nodes on level k i.e. level k can have maximum $2^k \times 2 = 2^{k+1}$ nodes. So if this property is true for any k, then it is also true for k+1. Hence proved.

Property 2 - The maximum number of nodes possible in a binary tree of height h is $2^h - 1$.

Proof : If we sum up the maximum number of nodes possible on each level then we can get the maximum number of nodes possible in the binary tree. First level is 0 and the last level is h-1(from definition of height) and at any level i the maximum number of nodes is 2^i. So the total number of nodes possible in a binary tree of height h is given by

$$n = \sum_{i=0}^{h-1} 2^i$$

$$= 1 + 2^1 + 2^2 + 2^3 + \ldots\ldots 2^{h-1} \text{ (geometric progression series)}$$

$$= \frac{2^{(h-1)+1} - 1}{2-1}$$

$$= 2^h - 1$$

Property 3 - The minimum number of nodes possible in a binary tree of height h is equal to h.

Proof : A tree will have minimum number of nodes if each level has minimum nodes. The minimum number of nodes possible on any level is 1 and there are total h levels (level 0 to level h-1). If we sum up the minimum number of nodes possible on each level then we can get the minimum number of nodes possible in the binary tree. So the minimum number of nodes possible in a binary tree of height h is equal to h.

The trees which have minimum number of nodes are called skew trees. These trees have h nodes and each tree has only one path. Some skew trees are shown below.

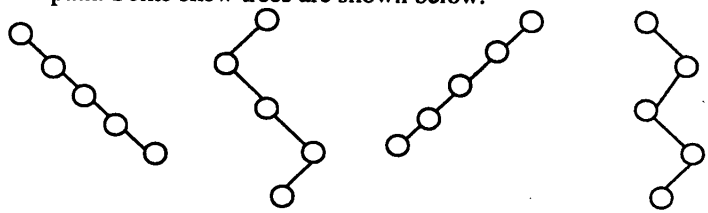


Figure 6.4 Skew Trees

Property 4 - If a binary tree contains n nodes, then its maximum height possible is n and minimum height possible is $\lceil \log_2(n+1) \rceil$.

Proof : There should be at least one element on each level, so the height cannot be more than n. A binary tree of height h can have maximum $2^h - 1$ nodes (from property 2). So the number of nodes will be less than or equal to this maximum value.

$n \leq 2^h - 1$

$2^h \geq n+1$

$h \geq \log_2(n+1)$ (Taking log of both sides)

$h \geq \lceil \log_2(n+1) \rceil$ (h is an integer)

So the minimum height possible is $\lceil \log_2(n+1) \rceil$.

Property 5 - In a non empty binary tree, if n is the total number of nodes and e is the total number of edges, then e = n-1

Proof : Every node in a binary tree has exactly one parent with the exception of root node. So if n is the total number of nodes then n-1 nodes have exactly one parent. There is only one edge between any child and its parent. So the total number of edges is n-1.

Another way of proving this property is by induction on n. If there is only one node then number of edges is 0, i.e. property is true for n=0 and induction base is proved. Suppose the property is true for a tree T whose number of nodes is k and number of edges is ed.

$k = ed + 1$ .....(i)

Now we insert one node in tree T and obtain tree T', this will increase one edge also. Now k' and ed' are the number of nodes and edges in tree T'

$k' = k+1$

$ed' = ed+1$

So we can write

$k = k' - 1$

$ed = ed' - 1$

Putting these values in equation (i) we get

$k' = ed' + 1$

So we have proved that if the property is true for a tree T with k nodes then it is also true for tree T' with k+1 nodes. Hence the property is proved.

Property 6 - For any non empty binary tree, if n_0 is the number of nodes with no child and n_2 is the number of nodes with 2 children, then $n_0 = n_2+1$.

Proof : Suppose n_1 is the number of nodes with 1 child. In a binary tree any node can have 0, 1 or 2 children so the total number of nodes n in a binary tree is given by-

$n = n_0 + n_1 + n_2$ .....(i)

If total number of edges is e, then from property 5 we know that

$e = n-1$ .....(ii)

Two edges emerge from the nodes which have two children, one edge emerges from the node that has one child and no edge emerges from the node that has no child, so the total number of edges e can be calculated as-

$e = 0 \times n_0 + 1 \times n_1 + 2 \times n_2$
$e = n_1 + 2n_2$
Now substitute the value of e from equation (ii)
$n - 1 = n_1 + 2n_2$
Now substitute the value of n from equation (i)
$n_0 + n_1 + n_2 - 1 = n_1 + 2n_2$
$n_0 = n_2 + 1$
Hence proved.

6.4 Strictly Binary Tree

A binary tree is a strictly binary tree if each node in the tree is either a leaf node or has exactly two children i.e. there is no node with one child.

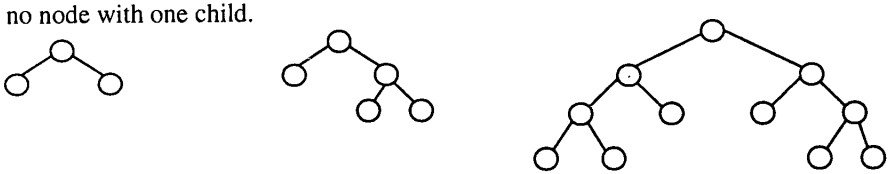


Figure 6.5 Strictly Binary Trees

The trees shown above are strictly binary trees because each node has either 0 or 2 children.

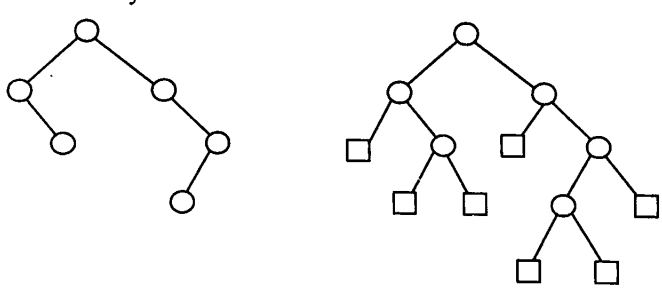
Property 7 - A strictly binary tree with n non leaf nodes has n+1 leaf nodes.
Proof : We can prove this property by induction on the number of non leaf nodes. If a binary tree consists only of the root node, then it has no non leaf node and one leaf node, so the property is true for n=0.

We will take a strictly binary tree with k non leaf nodes(k>0) and assume that this property is true for its left and right subtrees. Suppose the left subtree of this tree contains m non leaf nodes, so the right subtree will contain k-1-m non leaf nodes. Since the property is true for both these subtrees, left subtree will have m+1 leaf nodes and right subtree will have k-1-m+1 leaf nodes. So the total leaf nodes will be m+1+k-m i.e. k+1 nodes. Hence the property is true for the whole tree with k non leaf nodes.

Property 8 - A strictly binary tree with n leaf nodes always has 2n-1 nodes
Proof : From property 7, we know that if a strictly binary tree has n leaf nodes then it has n-1 non leaf nodes. The total number of nodes is the sum of leaf nodes and non leaf nodes i.e. n + n-1 = 2n-1 nodes.

6.5 Extended Binary Tree

If in a binary tree, each empty subtree(NULL link) is replaced by a special node then the resulting tree is extended binary tree or 2-tree. So we can convert a binary tree to an extended binary tree by adding special nodes to leaf nodes and nodes that have only one child. The special nodes added to the tree are called external nodes and the original nodes of the tree are internal nodes. The following figure shows a binary tree and the corresponding extended binary tree.



Binary tree Extended Binary tree

Figure 6.6

In the figure, external nodes are shown by squares and internal nodes by circles. We can see that all the external nodes are leaf nodes while the internal nodes are non leaf nodes.
The extended binary tree is a strictly binary tree i.e. each node has either 0 or 2 children.

The path length for any node is the number of edges traversed from that node to the root node. This path length is equal to the level number of that node. Path length of a tree is the sum of path lengths of all the nodes of the tree.

Internal path length of a binary tree is the sum of path lengths of all internal nodes which is equal to the sum of levels of all internal nodes. The internal path length of the tree given in figure 6.6 is-

$I = 0 + 1 + 1 + 2 + 2 + 3 = 9$

External path length of a binary tree is the sum of path lengths of all external nodes which is equal to the sum of levels of all external nodes. The external path length of the tree given in figure 6.6 is-

$E = 2 + 2 + 3 + 3 + 3 + 4 + 4 = 21$

The internal path length and external path length will be maximum when the tree is skewed(as in figure 6.4) and minimum when the tree is a complete binary tree(section 6.7). The following property shows the relation between internal and external path lengths.

Property 9 - In an extended binary tree, if E is the external path length, I is the internal path length and n is the number of internal nodes, then $E=I+2n$

Proof : This property can be proved by induction on the number of internal nodes. If tree contains only root then $E = I = n = 0$ and the theorem is correct for n=0. If n=1, then there is only one internal node which is the root node. The root node will have two children which are external nodes. So the internal path length is 0 and external path length is 2 i.e. E=2, I=0 and n=1. Hence the property is true for n=1 also.

Suppose we have a binary tree T that has k internal nodes. Let E and I be external and internal path lengths of this tree. Let A be an internal node in this tree such that both children of A are external nodes. Let p be the level of node A i.e. there are p edges from root to node A. We delete both children of node A from the tree. Node A is not an internal node now so the number of internal nodes is k-1. Suppose E′ and I′ are the external and internal path lengths of this resulting tree T′.

Two external nodes are deleted which are at level (p+1) so external path length decreases by 2(p+1) and node A becomes an external node so external path length increases by p.

$E' = E - 2(p+1) + p$ .....(1)

Node A is not an internal node now so internal path length decreases by p.

$I' = I - p$ .....(2)

Induction hypothesis : We assume that the property is true for the tree T′ that we obtain after deleting the two children of A.

$E' = I' + 2(k-1)$

Substituting the values of E′ and I′ from (1) and (2)-

$E - 2(p+1) + p = I - p + 2(k-1)$

On simplifying this equation we get-

$E = I + 2k$

So we have proved that if the property is true for tree T′ with k-1 internal nodes then it is also true for tree T with k internal nodes.

6.6 Full binary tree

A binary tree is full binary tree if all the levels have maximum number of nodes, i.e. if the height of tree is h, then it will have $2^h - 1$ nodes(see property 2). The following three trees are full binary trees of heights 2, 3, and 4 and contain 3, 7 and 15 nodes respectively.

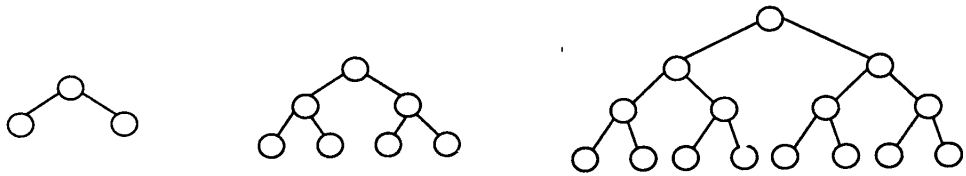


Figure 6.7 Full Binary Trees

The total number of nodes n in a full binary tree are $2^h - 1$, so its height $h = \log_2(n+1)$. Now suppose all nodes in a full binary tree are numbered from 1 to n starting from node on level 0, then nodes on level 1 and so on. Nodes on the same level are numbered from left to right. Root node is numbered 1.

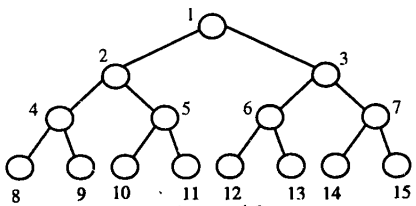


Figure 6.8

From the figure we can see that if the number of any node is k, then number of its left child is 2k, number of its right child is 2k+1 and number of its parent is floor(k/2).

6.7 Complete Binary Tree

A complete binary tree is a binary tree where all the levels have maximum number of nodes except possibly the last level. In the last level, number of nodes may range from 1 to 2^{h-1} and all these nodes are towards the left. The following three trees are complete binary trees.

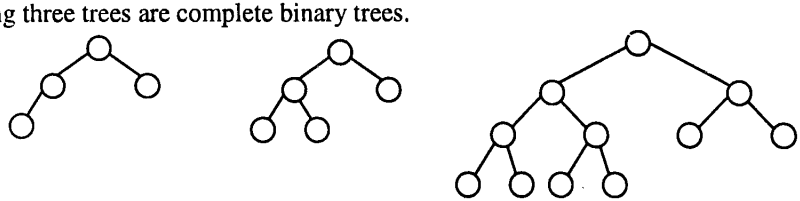


Figure 6.9 Complete Binary Trees

We can see that all the leaf nodes in a complete binary tree are on two adjacent levels (last level and second last level), and in the last level all the leaf nodes appear towards the left.

The height of complete binary tree with n nodes is $\lceil \log_2(n+1) \rceil$. If a complete binary tree contains 1000000 nodes then its height will be only 20.

Full binary tree can be considered as a special case of complete binary tree. The numbering of nodes and formula for finding out children and parent of a node in a complete binary tree is same as in full binary tree. In a complete binary tree, the last level may not have full nodes, so some nodes at the second last level may not have children. A node numbered k does not have a left child if $2k > n$ and it does not have a right child if $2k + 1 > n$.

The following property for complete binary trees summarizes these points.
If a node in a complete binary tree is assigned a number k, where $1 \leq k \leq n$, then
(i) If $k = 1$, then this node is root node. If $k > 1$, then its parent's number is floor(k/2).
(ii) If $2k > n$, then this node has no left child, otherwise the number of left child is 2k.
(iii) If $2k + 1 > n$, then this node has no right child, otherwise the number of right child is 2k+1.

Property 10 - If height of a complete binary tree is h, $h \geq 1$, then the minimum number of nodes possible is 2^{h-1} and maximum number of nodes possible is $2^h - 1$.
Proof : The number of nodes will be maximum when the last level also contains maximum nodes i.e. all levels are full, and so total nodes will be $2^h - 1$ from property 2.

The number of nodes will be minimum when the last level has only one node. In this case the total nodes will be -

total nodes in a full binary tree of height (h – 1) + one node

$= (2^{h-1} - 1) + 1 = 2^{h-1}$

(In some texts, strictly binary trees, full binary trees and complete binary trees are defined in a different way. In this book we'll use these terms as defined here.)

6.8 Representation of Binary Trees in Memory

Like lists, binary trees can also be implemented in two ways, one is array representation and the other is linked representation.

6.8.1 Array Representation of Binary trees

This is also called sequential representation or linear representation or contiguous representation or formula based representation of binary trees. Here we use a one-dimensional array to maintain the nodes of the binary tree. To decide the location of nodes inside the array, we utilize the numbering scheme described in full and complete binary trees.

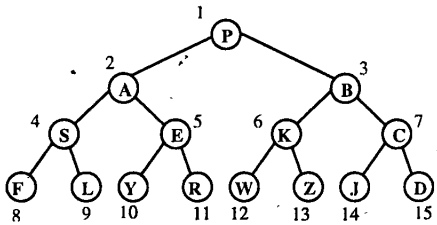


Figure 6.10

We'll store the nodes in the array named `tree` from index 1 onwards so if a node is numbered k then the data of this node is stored in `tree[k]`. The root node is stored in `tree[1]` and its left and right children in `tree[2]` and `tree[3]` respectively and so on. So the full binary tree in figure 6.10 can be stored in array as-

tree		P	A	B	S	E	K	C	F	L	Y	R	W	Z	J	D
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

We can easily extend this technique for other binary trees also. For this we consider any binary tree as a complete binary tree with some missing nodes. The nodes that are missing are also numbered and the array locations corresponding to them are left empty. For example-

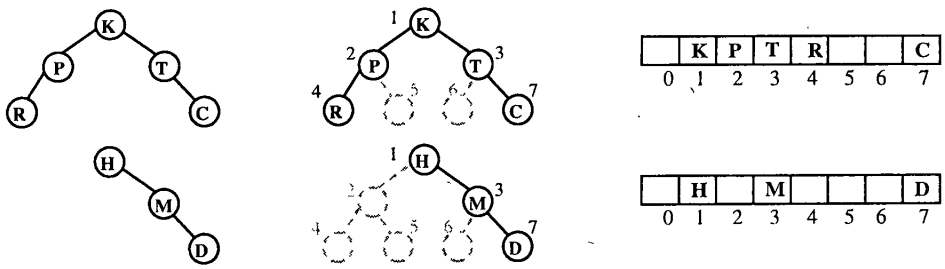


Figure 6.11

If a node is at index k in the array then its left child will be at index 2k and right child will be at index 2k+1. Its parent will be at index floor(k/2). We can see that lots of locations in the array are wasted if there are many missing nodes.

We have left the location 0 empty, it may be used for some other purpose. If we want to store nodes in the array starting from the index 0, then we can number the nodes from 0 to n-1. Root node is numbered 0 and

stored at index 0 in the array. Now if a node is at index k in the array then its left child will be at index 2k+1 and right child will be at index 2k+2. Its parent will be at index floor((k-1)/2).

The sequential representation is efficient from execution point of view because we can calculate the index of parent and index of the left and right children from the index of node. There is no need to have explicit pointers pointing to other nodes; the relationships are implicit.

We know that the maximum number of nodes possible in a binary tree of height h is $2^h - 1$ so the size of array needed is equal to $2^h - 1$. This size will be minimum if h is minimum, and it will be maximum if h is maximum. The height is minimum for complete binary trees which is $\lceil \log_2(n+1) \rceil$ so in this case size of array needed is-

$2^{\lceil \log_2(n+1) \rceil} - 1$.

The height is maximum(equal to n) for skewed binary trees, and so in this case size of array needed $2^n - 1$.

We have seen that lot of space is wasted if there are many missing nodes. The maximum wastage occurs in the case of a right-skewed binary tree of height h, it would require an array of size 2^n-1 out of which only n positions will be occupied. For example a right skewed tree having 20 nodes (height = 20) would require an array of size 1048575 but only 20 of these locations would be used. A complete binary tree of 20 nodes(height =5) would at most require an array of size 31. So this method is not very efficient in terms of space for trees other than complete and full binary trees.

Sequential representation is a static representation and the size of tree is restricted because of the limitation of array size. The size of array has to be known in advance and if array size is taken too small then overflow may occur and if array size is too large then space may be wasted.

Insertion and deletion of nodes requires lot of movement of nodes in the array which consumes lot of time, so it is suitable only for data that won't change frequently.

6.8.2 Linked Representation of Binary Trees

The linked representation overcomes the problems encountered in array representation. In this representation, explicit pointers are used to link the nodes of the tree. In array representation, the relationships between nodes were defined implicitly. In linked representation, we take three members in a node of the tree. First member is a pointer that stores the address of left child, second member is for data (this can be whole record), and third member is again a pointer that stores the address of right child.

The structure for tree node can be declared as-

```
struct node {
              struct node *lchild;
              char data;
              struct node  *rchild;
            };
```

This is a self referential structure where first and third members are structure pointers which point to left and right children of the node and second member `data` is information field of the node. If node has no left child then `lchild` should be NULL and if node has no right child then `rchild` should be NULL. Leaf nodes have no children so both `lchild` and `rchild` pointers will be NULL for them. Let us take a tree and see how it can be represented through linked representation.

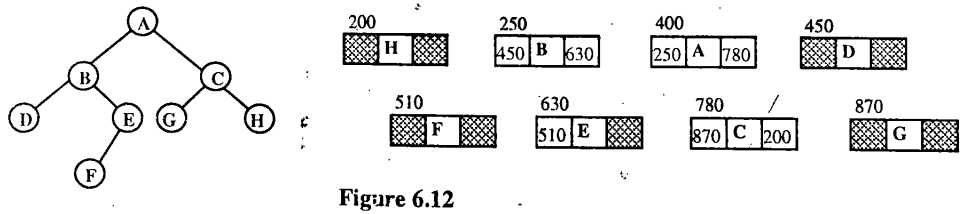


Figure 6.12

The nodes and their memory addresses are shown in the figure. For example the node A is located at address 400 and its left and right child pointers contain the addresses 250 and 780 which are the addresses of its left child B and right child C. For leaf nodes, the left and right pointers are NULL. We can see that the nodes of the

tree are scattered here and there in memory, but still they are connected to each other through the `lchild` and `rchild` pointers, which also maintain the hierarchical order of the tree. The logical view of the linked representation of the binary tree in figure 6.12 can be shown as-

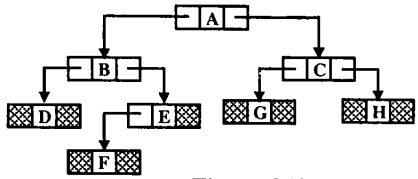


Figure 6.13

To access the nodes of the tree, we will maintain a pointer that will point to the root node of the tree.

```
struct node *root;
```

If the tree is empty then `root` will be NULL.

If we have a pointer `ptr` pointing to a node N, then `ptr->lchild` will point to left child of N and `ptr->rchild` will point to rchild of N. We can get a pointer to left child of root node by `root->lchild` and to the right child of root node by `root->rchild`.

We can move down the tree because we have pointers to children, but we can't find the parent of a given node. If we want this, we have to include a fourth member in the structure of node which is a pointer to the parent node. But most of the operations can be performed without the parent node information.

The linked representation uses dynamic memory allocation and so there is no restriction on the size of tree. Insertions and deletions are also not very time consuming as compared to sequential representation. The drawback of linked representation is that it requires more memory because of the pointers. Many of the pointers would be NULL but still they are required. Later we will see how we can use these null pointers also to store some useful information. The next property shows the number of null pointers in a binary tree.

Property 11 - If there are n nodes in a binary tree then the total number of null links will be $p = n + 1$.

Proof : Form property 5 we know that a tree with n nodes has n-1 edges. The number of nodes is n, so there will be 2n links and from these 2n links only n-1 links will have addresses of child nodes. So total null links will be $2n-(n-1) = n+1$.

We can prove this property by induction also. If n=1, only root node is there in the tree and it has two null links so property is true for n=1. If there are two nodes, then one will be the root node and the other will be the left or right child of root node. So there will be one null link of root node and two null links of child node i.e. total 3 null links for 2 nodes and hence the property is true for n=2 also.

Suppose property is true for a tree T with k nodes and p null links.

$p = k + 1$ .....(i)

We insert one more node in the tree and this will lead to removal of one null link but two new null links will appear, hence the total null links of the tree T will increase by 1. So the nodes and null links of new tree T' are k' and p' where $k' = k + 1$, $p' = p + 1$

We can write this as -

$k = k' - 1$

$p = p' - 1$

Putting these values in equation (i) we get

$p' = k' + 1$

So we have proved that if the property is true for a tree T with k nodes then it is also true for tree T' with k+1 nodes. Hence the property is proved.

6.9 Traversal in Binary Tree

Traversing a binary tree means visiting each node of the tree exactly once. Traversal of tree gives a linear order of the nodes, i.e. all nodes can be put in one line. We have done traversal of linked list and it was very simple there because a linked list is linear and to visit each node we just move from start to end. In a tree, nodes are arranged in hierarchical order and so there can be many ways in which these nodes can be visited.

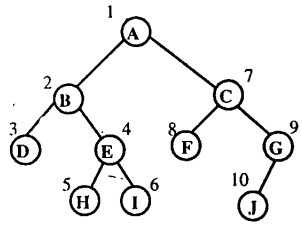
If we think of a binary tree recursively, then there are three main tasks for traversing it – visiting the root node, traversing its left subtree and traversing its right subtree. These three tasks can be performed in different orders, so we have 3! = 6 ways in which a tree can be traversed. If we name these tasks as N, L, R where visiting a node is N, traversing left subtree is L, traversing right subtree is R, then the 6 ways of traversal are NRL, NLR, LNR, LRN, RNL, RLN. If we follow the convention that left subtree is traversed before right subtree then we are left with only three traversals, which are NLR, LNR, LRN. These three are standard traversals and are given special names - preorder(NLR), inorder(LNR) and postorder(LRN). The prefix *pre* in preorder means visit the root first, prefix *in* in inorder means visit the root in between the subtrees and prefix *post* in postorder means visit the root at the end. These traversals can be defined recursively as-

Preorder Traversal (NLR)
1.Visit the root(N).
2.Traverse the left subtree of root in preorder(L).
3.Traverse the right subtree of root in preorder(R).

Inorder Traversal (LNR)
1. Traverse the left subtree of root in inorder(L).
2. Visit the root(N).
3. Traverse the right subtree of root in inorder(R).

Postorder Traversal (LRN)
1.Traverse the left subtree of root in postorder(L).
2.Traverse the right subtree of root in postorder(R).
3. Visit the root(N).

Let us take a binary tree and apply each traversal. The numbers written outside the nodes represent the sequence in which the nodes are visited in a particular traversal.

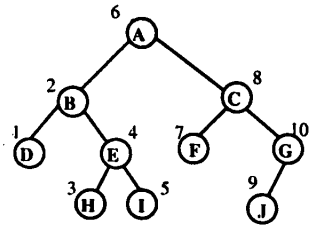


Preorder Traversal : A B D E H I C F G J

-- Visit **A**
-- Traverse left subtree of A in preorder
- Visit **B**
- Traverse left subtree of B in preorder
  - Visit **D**
- Traverse right subtree of B in preorder
  - Visit **E**
  - Traverse left subtree of E in preorder
    - Visit **H**
  - Traverse right subtree of E in preorder
    - Visit **I**

-- Traverse right subtree of A in preorder
- Visit **C**
- Traverse left subtree of C in preorder
  - Visit **F**
- Traverse right subtree of C in preorder
  - Visit **G**
  - Traverse left subtree of G in preorder
    - Visit **J**
  - Traverse right subtree of G in preorder
    - Empty

Figure 6.14 Preorder Traversal

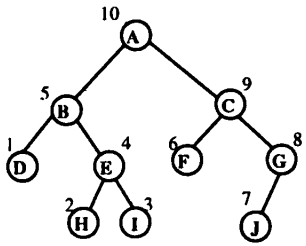


Inorder Traversal : D B H E I A F C J G

-- Traverse left subtree of A in inorder
- Traverse left subtree of B in inorder
  Visit **D**
- Visit **B**
- Traverse right subtree of B in inorder
  - Traverse left subtree of E in inorder
    - Visit **H**
  - Visit **E**
  - Traverse right subtree of E in inorder
    - Visit **I**

-- Visit **A**
-- Traverse right subtree of A in inorder
- Traverse left subtree of C in inorder
  - Visit **F**
- Visit **C**
- Traverse right subtree of C in inorder
  - Traverse left subtree of G in inorder
    - Visit **J**
  - Visit **G**
  - Traverse right subtree of G in inorder
    - Empty

Figure 6.15 Inorder Traversal



Postorder Traversal : D H I E B F J G C A

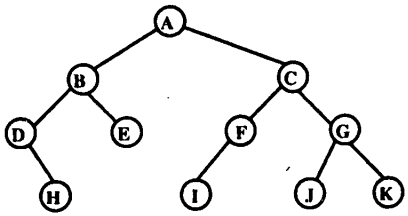
-- Traverse left subtree of A in postorder
- Traverse left subtree of B in postorder
  - Visit **D**
- Traverse right subtree of B in postorder
  - Traverse left subtree of E in postorder
    - Visit **H**
  - Traverse right subtree of E in postorder
    - Visit **I**
  - Visit **E**
- Visit **B**

-- Traverse right subtree of A in postorder
- Traverse left subtree of C in postorder
  - Visit **F**
- Traverse right subtree of C in postorder
  - Traverse left subtree of G in postorder
    - Visit **J**
  - Traverse right subtree of G in postorder
    - Empty
  - Visit **G**
- Visit **C**

-- Visit **A**

Figure 6.16 Postorder Traversal

Let us take another example of a binary tree and apply each traversal.



Preorder : A B D H E C F I G J K
Inorder : D H B E A I F C J G K
Postorder : H D E B I F J K G C A

Figure 6.17

The three traversals have been defined recursively, so they can be implemented using a stack. If we implement them recursively then an implicit stack is used and if we implement them non recursively then we have to use an explicit stack. We'll see both the implementations of all three traversals.

We assume that a binary tree already exists and is represented using the linked representation where the pointer `root` is a pointer to the root node of the tree. We will take an integer value as the information in each node of the tree, and visiting a node would mean printing this value.

The recursive functions for the three traversals are given below. All these functions are passed the address of the root.

```
void preorder(struct node *ptr)
{
        if(ptr==NULL)/*Base Case*/
                return;
        printf("%d  ",ptr->info);
        preorder(ptr->lchild);
        preorder(ptr->rchild);
}/*End of preorder()*/
void inorder(struct node *ptr)
{
        if(ptr==NULL)/*Base Case*/
                return;
        inorder(ptr->lchild);
        printf("%d  ",ptr->info);
        inorder(ptr->rchild);
}/*End of inorder()*/
void postorder(struct node *ptr)
{
        if(ptr==NULL)/*Base Case*/
                return;
        postorder(ptr->lchild);
        postorder(ptr->rchild);
        printf("%d  ",ptr->info);
}/*End of postorder()*/
```

6.9.1 Non recursive traversals for binary tree

If we want to write non recursive functions for traversals, then we will have to use an explicit stack. We assume that we have a stack, implemented using array and this stack will store the addresses of the nodes. The functions for implementing this stack are-

```
struct node *stack[MAX];
int top=-1;
void push_stack(struct node *item)
{
        if(top==(MAX-1))
        {
                printf("Stack Overflow\n");
                return;
        }
        top=top+1;
        stack[top]=item;
}/*End of push_stack()*/
struct node *pop_stack()
{
        struct node *item;
        if(top==-1)
        {
                printf("Stack Underflow....\n");
                exit(1);
        }
        item=stack[top];
```

```
        top=top-1;
        return item;
}/*End of pop_stack()*/
int stack_empty()
{
        if(top==-1)
                return 1;
        else
                return 0;
} /*End of stack_empty*/
```

In the explanation when we say push or pop a node, we would actually mean push or pop the address of that node.

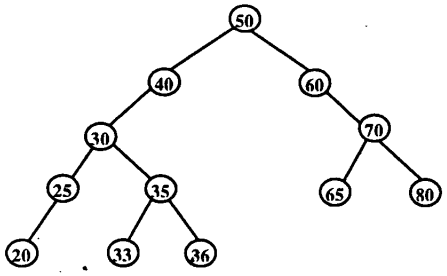


Figure 6.18

6.9.1.1 Preorder Traversal

The procedure for traversing a tree in preorder non recursively is-

1. Push the root node on the stack.
2. Pop a node from the stack.
3. Visit the popped node.
4. Push right child of visited node on stack
5. Push left child of visited node on stack.
6. Repeat steps 2,3,4,5 till the stack is not empty.

If we apply this process to the tree given in figure 6.18, the steps would be-

- Push 50
- Pop 50 : Visit **50**
  Push right and left child of 50 : Push 60 , Push 40
- Pop 40 : Visit **40**
  40 has no right child so push its left child : Push 30
- Pop 30 : Visit **30**
  Push right and left child of 30 : Push 35 , Push 25
- Pop 25 : Visit **25**
  25 has no right child so push its left child : Push 20
- Pop 20 : Visit **20**
  20 has no children so nothing is pushed
- Pop 35 : Visit **35**
  Push right and left child of 35 : Push 36, Push 33
- Pop 33 : Visit **33**
  33 has no children so nothing is pushed
- Pop 36 : Visit **36**
  36 has no children so nothing is pushed
- Pop 60 : Visit **60**
  60 has no left child so push its right child : Push 70
- Pop 70 : Visit **70**
  Push right and left child of 70 : Push 80, Push 65

- Pop 65 : Visit **65**
  65 has no children so nothing is pushed
- Pop 80 : Visit **80**
  80 has no children so nothing is pushed
- Stack empty

Preorder traversal : 50 40 30 25 20 35 33 36 60 70 65 80

The function for non recursive preorder traversal is-

```
void nrec_pre(struct node *root)
{
        struct node *ptr = root;
        if(ptr==NULL)
        {
                printf("Tree is empty\n");
                return;
        }
        push_stack(ptr);
        while(!stack_empty())
        {
                ptr = pop_stack();
                printf("%d  ",ptr->info);
                if(ptr->rchild!=NULL)
                        push_stack(ptr->rchild);

                if(ptr->lchild!=NULL)
                        push_stack(ptr->lchild);
        }
        printf("\n");
}/*End of nrec_pre*/
```

6.9.1.2 Inorder Traversal

The procedure for traversing a tree in inorder non recursively is-

1. Initially `ptr` is assigned address of root node.

2. Move along leftmost path rooted at the node pointed by `ptr`, pushing all the nodes in the path on the stack. Stop when we reach the leftmost node i.e. a node which has no left child, it is not pushed on the stack. Now `ptr` points to this leftmost node.

3. If node pointed by `ptr` has no right subtree, then visit it and pop another one from stack. Now keep on popping and visiting the nodes till a node is popped that has a right subtree. Now `ptr` points to this node that has a right subtree. If the stack becomes empty then the traversal is finished.

4. Visit the node pointed by `ptr`, and now `ptr` is assigned the address of its right child.
   Go to step 2.

Since we move along leftmost path and push nodes, we know that whenever we pop a node, its left subtree traversal is finished. So a node is visited just after popping it from the stack.

If we apply this process to the tree given in figure 6.18 the steps would be-

(i) Move along the leftmost path rooted at 50
- Push 50
- Push 40
- Push 30
- Push 25
- 20 is leftmost node, it has no right subtree : Visit **20** and pop
- 25 popped : it has no right subtree : Visit **25** and pop
- 30 popped : it has right subtree : Visit **30** and move to its right subtree

(ii) Move along leftmost path rooted at 35
- Push 35
- 33 is leftmost node, it has no right subtree : Visit **33** and pop
- 35 popped : it has right subtree : Visit **35** and move to its right subtree

(iii) Move along leftmost path rooted at 36
- 36 is leftmost node, it has no right subtree : Visit **36** and pop
- 40 popped: it has no right subtree: Visit **40** and pop
- 50 popped: it has right subtree: Visit **50** and move to its right subtree

(iv) Move along leftmost path rooted at 60
- 60 is leftmost node, it has a right subtree : Visit **60** and move to its right subtree

(v) Move along leftmost path rooted at 70
- Push 70
- 65 is leftmost node, it has no right subtree : Visit **65** and pop
- 70 popped : it has right subtree : Visit **70** and move to its right subtree

(vi) Move along leftmost path rooted at 80
- 80 is leftmost node, it has no right subtree : Visit **80**, Stack empty

Inorder traversal : 20 25 30 33 35 36 40 50 60 65 70 80

The function for non-recursive inorder traversal is -

```
void nrec_in(struct node *root)
{
        struct node *ptr=root;
        if(ptr==NULL)
        {
                printf("Tree is empty\n");
                return;
        }
        while(1)
        {
               while(ptr->lchild!=NULL)
                {
                        push_stack(ptr);
                        ptr = ptr->lchild;
                }

                while(ptr->rchild==NULL)
                {
                        printf("%d  ",ptr->info);
                        if(stack_empty())
                                return;
                        ptr = pop_stack();
                }
                printf("%d  ",ptr->info);
                ptr = ptr->rchild;
        }
        printf("\n");
}/*End of nrec_in()*/
```

6.9.1.3 Postorder Traversal

In inorder traversal, we had to visit only the left subtree before visiting the node, but in postorder we have to visit both left and right subtrees before visiting the node. We'll make changes in the procedure of inorder to make sure the right subtree is also traversed before visiting the node.

In step 4 of inorder, we visited the node that has right child, but in postorder we will push it on the stack. So nodes that have both left and right subtrees are pushed on the stack twice, nodes that have only one subtree are pushed on the stack once and leaf nodes are never pushed. The procedure for traversing a tree in postorder non recursively is -

1. Initially `ptr` is assigned address of root node.

2. Move along leftmost path rooted at node `ptr` pushing all the nodes in the path on the stack. Stop when we reach the leftmost node i.e. a node which has no left child, it is not pushed on the stack. Now `ptr` points to this leftmost node.

3. If `ptr` has no right subtree or its right subtree has been traversed then visit the node and pop another one from stack. Now keep on popping and visiting the nodes till a node is popped that has a right subtree which has not been traversed. Now `ptr` points to this node that has an untraversed right subtree. If the stack becomes empty then the traversal is finished.

4. Push the node pointed to by `ptr`, and now `ptr` is assigned the address of its right child.
Go to step 2.

Since we move along leftmost path and push nodes, we know that whenever we pop a node, its left subtree traversal is finished. In inorder we could visit a node just after popping it from the stack, but here we have to traverse the right subtree also. So here the node is again pushed on stack if it has untraversed right subtree.

If we apply this process to the tree given in figure 6.18 the steps would be-

(i) Move along leftmost path rooted at 50
- Push 50
- Push 40
- Push 30
- Push 25
- 20 is leftmost node, it has no right subtree : Visit **20** and pop
- 25 popped : it has no right subtree : Visit **25** and pop
- 30 popped : it has right subtree which has not been traversed : Push 30 and move to its right subtree

(ii) Move along leftmost path rooted at 35
- Push 35
- 33 is leftmost node, it has no right subtree : Visit **33** and pop
- 35 popped : it has a right subtree which has not been traversed : Push 35 and move to its right subtree

(iii) Move along leftmost path rooted at 36
- 36 is leftmost node, it has no right subtree : Visit **36** and pop
- 35 popped : it has right subtree which has been traversed : Visit **35** and pop
- 30 popped : it has right subtree which has been traversed : Visit **30** and pop
- 40 popped : it has no right subtree : Visit **40** and pop
- 50 popped: it has right subtree which has not been traversed : Push 50 and move to its right subtree

(iv) Move along leftmost path rooted at 60
- 60 is leftmost node, it has right subtree which has not been traversed : Push 60 and move to its right subtree

(v) Move along leftmost path rooted at 70
- Push 70
- 65 is leftmost node, it has no right subtree : Visit **65** and pop
- 70 popped : it has right subtree which has not been traversed : Push 70 and move to its right subtree

(vi) Move along leftmost path rooted at 80
- 80 is leftmost node, it has no right subtree : Visit **80** and pop
- 70 popped : it has right subtree which has been traversed : Visit **70** and pop
- 60 popped : it has right subtree which has been traversed : Visit **60** and pop
- 50 popped : it has right subtree which has been traversed : Visit **50**, stack empty

20 25 33 36 35 30 40 65 80 70 60 50

While writing the function for postorder we will use a secondary pointer q to know whether the right subtree has been traversed or not.

The function for non recursive postorder traversal is-

```
void nrec_post(struct node *root)
{
        struct node *q, *ptr = root;
        if(ptr==NULL)
        {
                printf("Tree is empty\n");
                return;
        }
        q = root;
        while(1)
        {
                while(ptr->lchild!=NULL)
                {
                        push_stack(ptr);
                        ptr=ptr->lchild;
                }
                while(ptr->rchild==NULL || ptr->rchild==q)
                {
                        printf("%d  ",ptr->info);
                        q = ptr;
                        if(stack_empty())
                                return;
                        ptr = pop_stack();
                }
                push_stack(ptr);
                ptr = ptr->rchild;
        }
        printf("\n");
}/*End of nrec_post()*/
```

If the right subtree of ptr has already been traversed, then the node visited recently would be the root of right subtree of ptr. The pointer q is used to store the address of recently visited node.

6.9.2 Level order traversal

In level order traversal, the nodes are visited from top to bottom and from left to right. Initially we visit the root node at level 0, then we visit all the nodes of level 1, and then all nodes of level 2 and so on till the last level. The nodes of a particular level are visited from left to right. Let us take a tree and see its level order traversal.

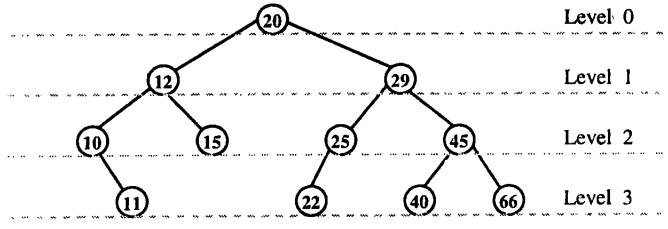


Figure 6.19 Level Order Traversal

The level order traversal of this tree is-
20 12 29 10 15 25 45 11 22 40 66

This traversal can be implemented using a queue that will store the addresses of the nodes. The procedure for level order traversal is-

1. Insert root node in the queue.
2. Delete a node from the front of queue and visit it.
3. Insert the left child of visited node in the queue at the end.
4. Insert the right child of the visited node in the queue at the end.
5. Repeat steps 2, 3, 4 till the queue is not empty.

If we apply the above procedure for the tree in figure 6.19 the steps would be-

- Insert 20
- Delete 20 : Visit **20**
- Insert 12 and 29
- Delete 12 : Visit **12**
- Insert 10 and 15
- Delete 29 : Visit **29**
- Insert 25 and 45
- Delete 10 : Visit **10**
- Insert 11
- Delete 15 : Visit **15**
- Delete 25 : Visit **25**
- Insert 22
- Delete 45 : Visit **45**
- Insert 40 and 66
- Delete 11 : Visit **11**
- Delete 22 : Visit **22**
- Delete 40 : Visit **40**
- Delete 66 : Visit **66**

```
struct node *queue[MAX];
int front=-1,rear=-1;
void insert_queue(struct node *item)
{
        if(rear==MAX-1)
        {
                printf("Queue Overflow\n");
                return;
        }
        if(front==-1)  /*If queue is initially empty*/
                front=0;
        rear=rear+1;
        queue[rear]=item ;
}/*End of insert()*/
struct node *del_queue()
{
        struct node *item;
        if(front==-1 || front==rear+1)
        {
                printf("Queue Underflow\n");
                return 0;
        }
        item=queue[front];
        front=front+1;
        return item;
}/*End of del_queue()*/
int queue_empty()
{
        if(front==-1 || front==rear+1)
                return 1;
        else
                return 0;
}
```

The function for level order traversal can be written as-

```
void level_trav(struct node *root)
{
        struct node *ptr = root;
        if(ptr==NULL)
        {
                printf("Tree is empty\n");
                return;
        }
        insert_queue(ptr);
        while(!queue_empty())/*Loop until queue is not empty*/
        {
                ptr=del_queue();
                printf("%d ",ptr->info);
                if(ptr->lchild!=NULL)
                        insert_queue(ptr->lchild);
                if(ptr->rchild!=NULL)
                        insert_queue(ptr->rchild);
        }
        printf("\n");
}/*End of level_tray()*/
```

6.9.3 Creation of binary tree from inorder and preorder traversals

The inorder, preorder or postorder traversals of different binary trees may be same, so if we are given a single traversal we can't construct a unique binary tree. However if we know the inorder and preorder traversals or inorder and postorder traversals then we can construct a unique binary tree. Note that inorder traversal is necessary for drawing the tree; we can't draw a tree from only preorder and postorder traversals. Suppose we are given a preorder XY and postorder YX then it is clear that X is the root but we can't find out whether Y is left child or right child of X.

The procedure for constructing a binary tree from inorder and preorder traversals is-

(i) In preorder traversal, first node is the root node, so we get the root node by taking the first node of preorder. Now all remaining nodes form the left and right subtrees of the root. To divide them into left and right subtrees we look at the inorder traversal.

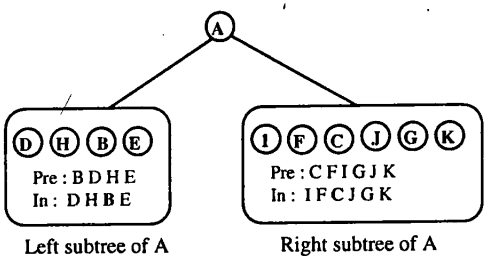
(ii) In inorder traversal, root is in the middle, and nodes to the left of the root node are nodes of left subtree of node, and nodes to the right of the root node are nodes of right subtree of root node. Since we know the root node(first node of preorder), we can separate the nodes of left and right subtrees.

(iii) Now we can follow the same procedure for both left and right subtrees till we get an empty subtree or a single node in subtree. Inorder and preorder for these subtrees can be obtained from the inorder and preorder of the whole tree.

Let us construct a binary tree from given preorder and inorder traversals.

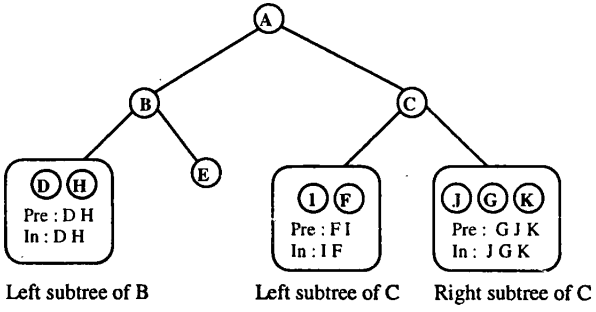
Preorder : A B D H E C F I G J K
Inorder : D H B E A I F C J G K

In preorder traversal, first node is the root node. Hence A is the root of the binary tree. From inorder, we see that nodes to the left of root node A are nodes D, H, B, E so these nodes form the left subtree of A, similarly nodes I, F, C, J, G, K form the right subtree of A since they are to the right of A.



Left subtree of A :
From preorder we get node B as root
From inorder we get nodes D, H in left subtree of B, and node E in right subtree of B.

Right subtree of A :
From preorder we get C as the root
From inorder we get nodes I, F in left subtree of C, and nodes J, G, K in right subtree of C.



Left subtree of B :
From preorder we get D as the root
From inorder we get empty left subtree of D, and node H in right subtree of D.

Left subtree of C :
From preorder we get F as the root
From inorder we get node I in left subtree of F, and empty right subtree of F.

Right subtree of C :
From preorder we get G as the root
From inorder we get node J in left subtree of G, and node K in right subtree of G.

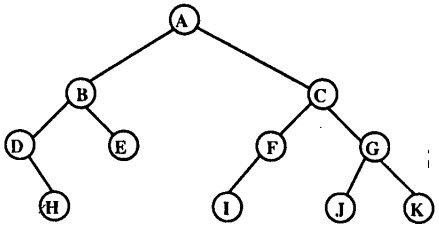


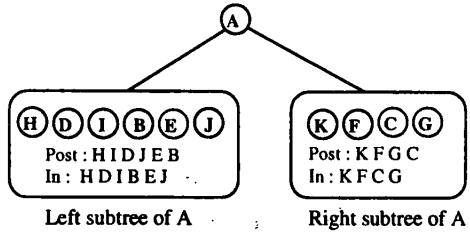
Figure 6.20

6.9.4 Creation of binary tree from inorder and postorder traversals

We can create a binary tree from postorder and inorder traversals using a similar procedure as described in previous section. The only difference is that here we get the root node by taking the last node of postorder traversal. Let us construct a binary tree from given postorder and inorder traversals.

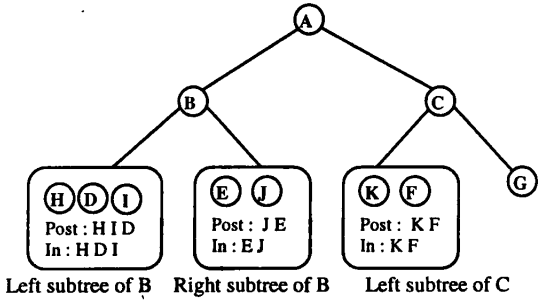
Postorder : H I D J E B K F G C A
Inorder : H D I B E J A K F C G

Node A is last node in postorder traversal, so it will be the root of the tree. From inorder, we see that nodes to the left of the root node A are nodes H, D, I, B, E, J, so these nodes form the left subtree of A, similarly nodes K, F, C, G form the right subtree of A since they are to the right of A.



Left subtree of A :
From postorder we get node B as root.
From inorder we get nodes H, D, I in left subtree of B, and nodes E, J in right subtree of B.

Right subtree of A :
From postorder we get node C as root.
From inorder we get nodes K, F in left subtree of C, and node G in right subtree of C.



Left subtree of B :
From postorder we get node D as root.
From inorder we get nodes H in left subtree of D, and node I in right subtree of D.

Right subtree of B :
From postorder we get node E as root.
From inorder we get empty left subtree of E, and node J in right subtree of E.

Left subtree of C :
From postorder we get node F as root.
From inorder we get node K in left subtree of F, and empty right subtree of F.

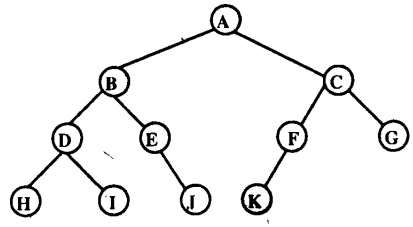


Figure 6.21

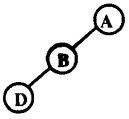
Now we'll see a quicker method of creating the tree from preorder and inorder traversal. In preorder traversal, scan the nodes one by one and keep inserting them in the tree. In inorder traversal, underline the nodes which have been inserted. To insert a node in its proper position in the tree, we will look at that node in the inorder traversal and insert it according to its position with respect to the underlined nodes.

Preorder : A B D G H E I C F J K
Inorder : G D H B E I A C J F K

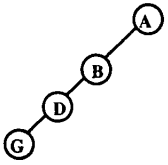
Insert A:
G D H B E I A C J F K
A is the first node in preorder, hence it is root of the tree.

Insert B :
G D H *B* E I A C J F K
B is to left of A, hence it is left child of A.

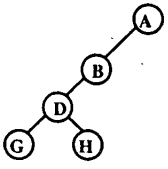
Insert D :
G *D* H B E I A C J F K
D is to the left of B, hence D is left child of B.



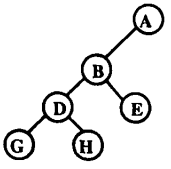
Insert G :
G D H B E I A C J F K
G is to the left of D, hence G is left child of D.



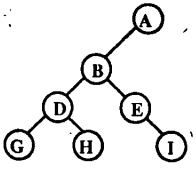
Insert H :
G D *H* B E I A C J F K
H is to the left of B and right of D, hence H is right child of D.



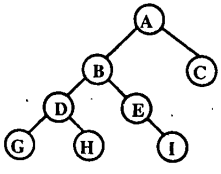
Insert E :
G D H B *E* I A C J F K
E is to the left of A and right of B, hence E is right child of B.



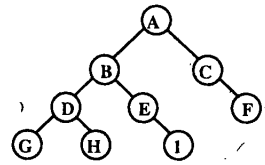
Insert I :
G D H B E *I* A C J F K
I is to the left of A and right of E, hence I is right child of E.



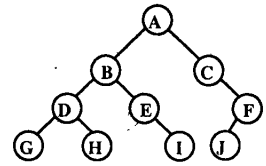
Insert C :
G D H B E I A *C* J F K
C is to the right of A, hence C is right child of A.



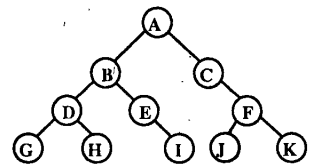
Insert F :
G D H B E I A C J *F* K
F is to the right of C, hence F is right child of C.



Insert J :
G D H B E I A C *J* F K
J is to the right of C and to left of F, hence J is left child of F.



Insert K :
G D H B E I A C J F *K*
K is to the right of F, hence K is right child of F.



Creation of tree from postorder and inorder by this method is same as creation of tree from inorder and preorder. The only difference is that in postorder we will start scanning the nodes from the right side i.e. the last node in the postorder will be inserted first and first node will be inserted at last.

The function for constructing a tree from inorder and preorder is given next. The inorder and preorder traversals are stored in linked lists. Structure for nodes of tree is declared as `struct treenode` and structure for nodes of list is declared as `struct listnode`.

```
struct treenode *construct(struct listnode *inptr, struct listnode *preptr,int num)
{
        struct treenode *temp;
        struct listnode *q;
        int i,j;
        if(num==0)
                return NULL;
        temp=(struct treenode *)malloc(sizeof(struct treenode));
        temp->info=preptr->info;
        temp->lchild = NULL;
        temp->rchild = NULL;
        if(num==1)/*if only one node in tree*/
                return temp;
        q = inptr;
        for(i=0; q->info != preptr->info; i++)
                q = q->next;
        /*Now q points to root node in inorder list and number of nodes in its
         left subtree is i*/
        /*For left subtree*/
        temp->lchild = construct(inptr, preptr->next, i);
        /*For right subtree*/
        for(j=1;j<=i+1;j++)
                preptr=preptr->next;
        temp->rchild =.construct(q->next, preptr, num-i-1);
        return temp;
}/*End of construct()*/
```

The function for constructing a tree from inorder and postorder is-

```
struct treenode *construct(struct listnode *inptr,struct listnode *postptr, int num)
{
        struct treenode *temp;
        struct listnode *q, *ptr;
```

```
        int i,j;
        if(num==0)
                return NULL;
        ptr=postptr;
        for(i=1; i<num; i++)
                ptr = ptr->next;
        /*Now ptr points to last node of postorder which is root*/
        temp=(struct treenode *)malloc(sizeof(struct treenode));
        temp->info=ptr->info;
        temp->lchild = NULL;
        temp->rchild = NULL;
        if(num==1)/*if only one node in tree*/
                return temp;
        q=inptr;
        for(i=0; q->info!=ptr->info; i++)
                q = q->next;
        /*Now i denotes the number of nodes in left subtree
        and q points to root node in inorder list*/
        /*For left subtree*/
        temp->lchild = construct(inptr, postptr, i);
        /*For right subtree*/
        for(j=1; j<=i; j++)
                postptr = postptr->next;
        temp->rchild = construct(q->next, postptr, num-i-1);
        return temp;
}/*End of construct()*/
```

6.10 Height of Binary tree

The recursive function to find out the height of binary tree is given below. It is passed the root of the binary tree and it returns the height of that tree.

```
int height(struct node *ptr)
{
        int h_left,h_right;
        if(ptr == NULL) /*Base Case*/
                return 0;
        h_left = height(ptr->lchild);
        h_right = height(ptr->rchild);
        if(h_left > h_right)
                return 1 + h_left;
        else
                return 1 + h_right;
}/*End of height()*/
```

The height of the empty tree is 0 and this serves as the base case. In the recursive case, we can find out the height of a tree by adding 1 to the height of its left or right subtree(whichever is more). For example suppose we have to find out the height of the given tree-

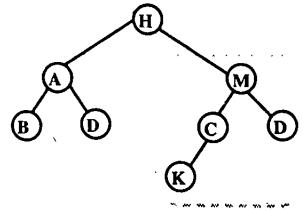
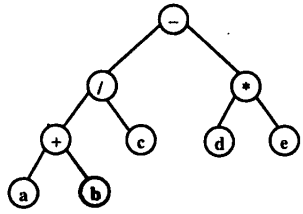


Figure 6.22

The heights of left and right subtrees rooted at H are 2 and 3 respectively. So the height of this tree is can be obtained by adding 1 to height of right subtree, and hence the height of this tree is 4. The height of left and right subtrees can be found out in the same way.

6.11 Expression tree

Any algebraic expression can be represented by a tree in which the non leaf nodes are operators and leaf nodes are the operands. Almost all arithmetic operations are unary or binary so the expression trees are generally binary trees. The left child represents the left operand while the right child represents the right operand. In the case of unary minus operator, the node will have only one child. Every algebraic expression represents a unique tree. Let us take an algebraic expression and the corresponding expression tree.



We can see that the leaf nodes are the variables or constants and all the non leaf nodes are the operators. The parentheses of the algebraic expression do not appear in the tree but the tree retains the purpose of parentheses, for example the + operator is applied to operands a and b and then the / operator is applied to a+b and c. Now we'll take an expression tree and apply each traversal.

Algebraic expression : (a – b * c) / (d + e / f)

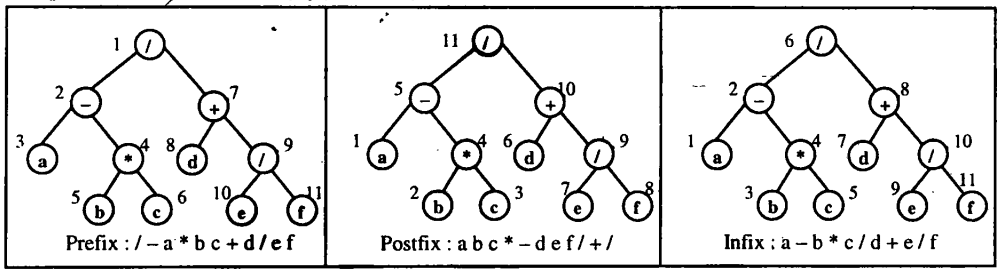


Figure 6.24

Preorder Traversal : / – a * b c + d / e f
Inorder Traversal : a – b * c / d + e / f
Postorder Traversal : a b c * – d e f / + /

The preorder and postorder traversals give the corresponding **prefix** and postfix expressions of the given algebraic expression. The inorder traversal gives us an **expression in which** the order of operator and operands are same as in the given algebraic expression but without **the parentheses**. We can get a parenthesized infix expression by following the given procedure-

- If the node is a leaf node (operand)
  Print the contents of the node
- If the node is a non leaf node (operator)
  Print ' ('
  Traverse the left subtree
  Print the contents of the node
  Traverse the right subtree
  Print ') '

Starting from the root node, the above recursive procedure will give us this expression-
((a – (b * c)) / (d + (e / f)))

This expression has many surplus pair of parentheses, but it exactly represents the algebraic expression (a – b * c) / (d + e / f).

We can easily construct an expression tree using either the postfix expression or prefix expression. We have already seen the procedure for converting an infix expression to postfix or prefix expression(in chapter 4). Now let us see the procedure for converting a postfix expression to an expression tree. This procedure uses a stack and is somewhat similar to the procedure of evaluating a postfix expression. The postfix expression is scanned from left to right, and if the symbol scanned is an operand then we allocate a tree node and push its pointer on the stack. If the symbol scanned is an operator, then we pop pointers to two nodes N1 and N2 from the stack and then create a new tree by making the scanned operator as the root with N1 and N2 as the children. If pointer to N1 is popped first then it will be the right child and N2 will be the left child, otherwise N2 will be the right child and N1 will be the left child. The figure 6.25 shows how we can construct the tree given in the example using the following postfix expression.

a b c * – d e f / + /

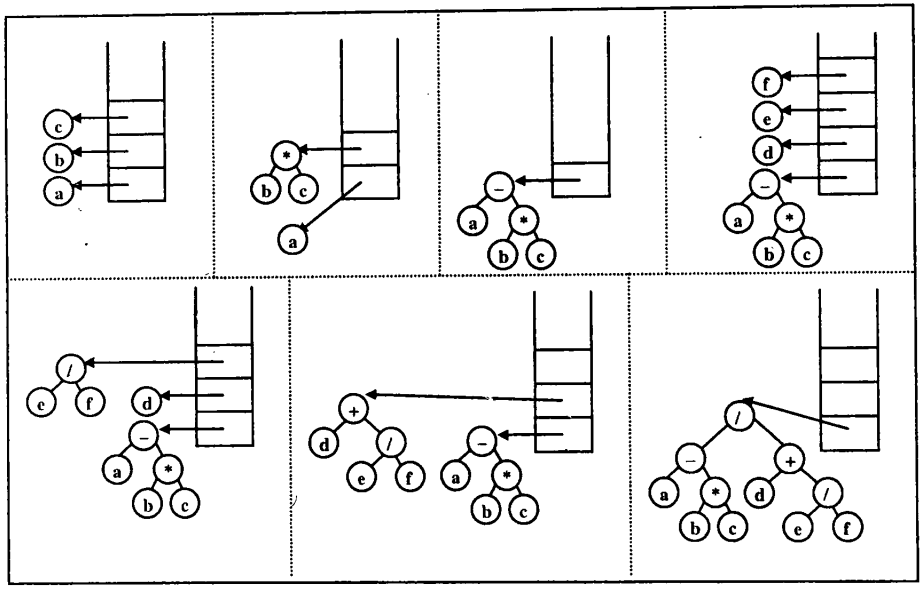


Figure 6.25

6.12 Binary Search Tree

One of the important uses of binary trees is in searching. Binary search trees are binary trees that are specially organized for the purpose of searching. An element can be searched in average O(log N) time where N is the number of nodes. In binary search tree, a key is associated with each node. We can define binary search trees recursively as-

A binary search tree is a binary tree that may be empty and if it is not empty then it satisfies the following properties-

1. All the keys in the left subtree of root are less than the key in the root.
2. All the keys in the right subtree of root are greater than the key in the root.
3. Left and right subtrees of root are also binary search trees.

We have assumed that all the keys of binary search tree are distinct, although there can be binary search tree with duplicates and in that case the operations explained here have to be changed.

For simplicity we assume that the data part contains only an integer value which also serves as the key. The following trees are examples of binary search trees.

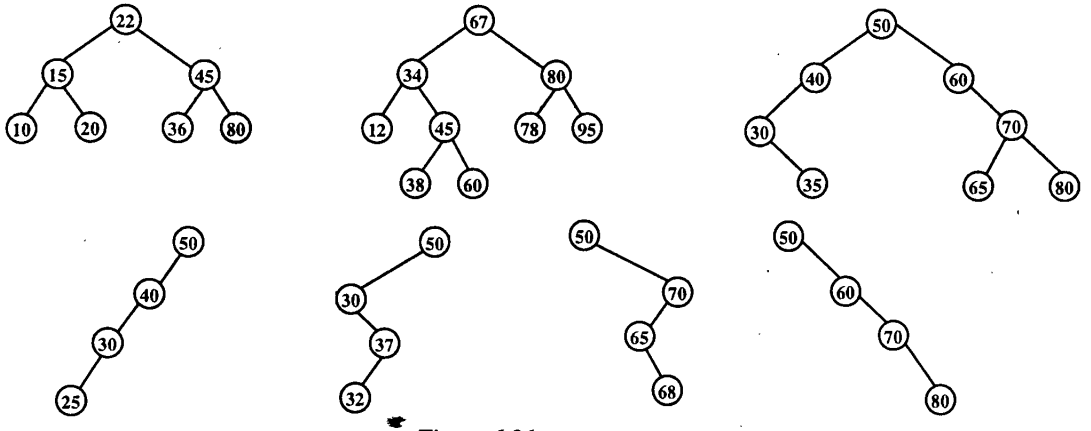


Figure 6.26

In all these trees we can see that for each node N in the tree, all the keys in left subtree of node N are smaller than key of node N and all the keys in right subtree of node N are greater than the key of node N.

6.12.1 Traversal in Binary Search Tree

Binary search tree is a binary tree so the traversal methods given for binary trees apply here also.

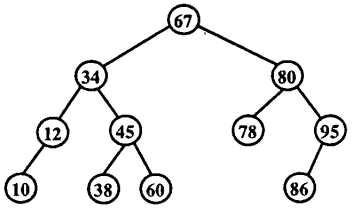


Figure 6.27

The different traversals for the binary search tree in figure 6.27 are-

Preorder : 67 34 12 10 45 38 60 80 78 95 86
Inorder : 10 12 34 38 45 60 67 78 80 86 95
Postorder : 10 12 38 60 45 34 78 86 95 80 67
Level order : 67 34 80 12 45 78 95 10 38 60 86

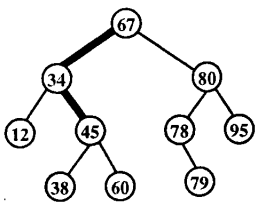
Note that the inorder traversal of a binary search tree gives us all keys of that tree in ascending order.

6.12.2 Searching in a Binary Search Tree

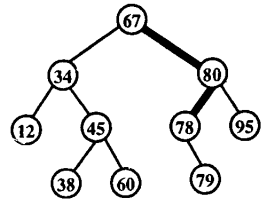
We start at the root node and move down the tree, and while descending when we encounter a node, we compare the desired key with the key of that node and take appropriate action.

1. If the desired key is equal to the key in the node then the search is successful.
2. If the desired key is less than the key of the node then we move to left child.
3. If the desired key is greater than the key of the node then we move to right child.

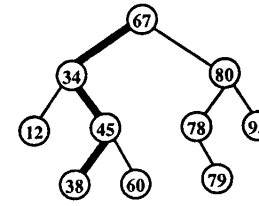
In the process, if we reach a NULL left child or NULL right child then the search is unsuccessful i.e. desired key is not present in the tree.



Search 45
45 < 67, move to left child
45 > 34, move to right child
45 found



Search 77
77 > 67, move to right child
77 < 80, move to left child
77 < 78, move to left child
NULL left child
77 not present



Search 38
38 < 67, move to left child
38 > 34, move to right child
38 < 45, move to left child
38 found

Figure 6.28 Searching in a Binary Search Tree

The function for searching a node non recursively is given below. This function is passed the address of the root of the tree and the key to be searched. It returns address of the node having the desired key or NULL if such a node is not found.

```
struct node *search_nrec(struct node *ptr, int skey)
{
        while(ptr!=NULL)
        {
                if(skey < ptr->info)
                        ptr = ptr->lchild; /*Move to left child*/
                else if(skey > ptr->info)
                        ptr = ptr->rchild; /*Move to right child*/
                else    /*skey found*/
                        return ptr;
        }
        return NULL; /*skey not found*/
}/*End of search_nrec()*/
```

The search process can also be described recursively, To search a key in the tree, first the key is compared with the key in the root node. If the key is found there, then the search is successful. If key is less than the key in the root, then the search is performed in left subtree because we know that all keys less than root are stored in left subtree. Similarly if key is greater than the key in the root then the search is performed in the right subtree. In this way the search is carried out recursively. The recursive process stops when we reach the base case of recursion. Here we have two base cases, first when we find the desired key(successful search) and second when we reach a NULL subtree (unsuccessful search). The recursive function for searching a node is given below. This function is passed the address of root of the tree and the value of the key to be searched.

```
struct node *search(struct node *ptr,int skey)
{
        if(ptr==NULL)
        {
                printf("key not found\n");
                return NULL;
        }
        else if(skey < ptr->info)  /*search in left subtree*/
                return search(ptr->lchild, skey);
        else if(skey > ptr->info)  /*search in right subtree*/
                return search(ptr->rchild, skey);
        else  /*skey found*/
                return ptr;
}/*End of search()*/
```

Searching in binary search tree is efficient because we just have to traverse a branch of the tree and not all the nodes sequentially as in linked list. The running time is O(h) where h is the height of the tree.

6.12.3 Finding nodes with Minimum and Maximum key

The last node in the leftmost path starting from root is the node with the smallest key. To find this node we start at the root and move along the leftmost path until we get a node with no left child. You can verify it by seeing the binary search trees given in the example. The non recursive and recursive functions to find this node are given below-

```
struct node *min_nrec(struct node *ptr)
{
        if(ptr!=NULL)
                while(ptr->lchild!=NULL)
                        ptr = ptr->lchild;
        return ptr;
}/*End of min_nrec()*/
struct node *Min(struct node *ptr)
{
        if(ptr==NULL)
                return NULL;
        else if(ptr->lchild==NULL)
                return ptr;
        else
                return Min(ptr->lchild);
}/*End of Min()*/ /
```

These functions will be called with the address of the root.

The last node in the rightmost path starting from root is the node with the largest key. To find this node we start at the root and move along the rightmost path until we get a node with no right child. The non recursive and recursive functions to find this node are given below-

```
struct node *max_nrec(struct node *ptr)
{
        if(ptr!=NULL)
                while(ptr->rchild!=NULL)
                        ptr = ptr->rchild;
        return ptr;
}/*End of max_nrec()*/
struct node *Max(struct node *ptr)
{
        if(ptr==NULL)
                return NULL;
        else if(ptr->rchild==NULL)
                return ptr;
        else
                return Max(ptr->rchild);
}/*End of Max()*/
```

Both these operations run in O(h) time, where h is the height of the tree.

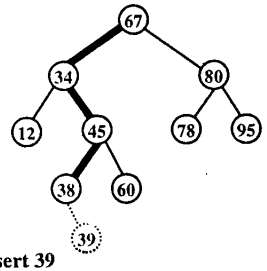
6.12.4 Insertion in a Binary Search Tree

For inserting a new node, first the key is searched and if it is not found in the tree then it is inserted at the place where search terminates.

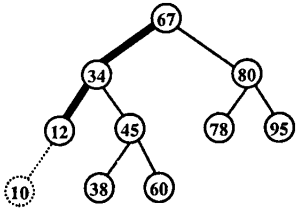
We start at the root node and move down the tree and while descending when we encounter a node we compare the key to be inserted with the key of that node and take appropriate action.

1. If the key to be inserted is equal to the key in the node then there is nothing to be done as duplicate keys are not allowed.
2. If the key to be inserted is less than the key of the node then we move to left child.
3. If the key to be inserted is greater than the key of the node then we move to right child.

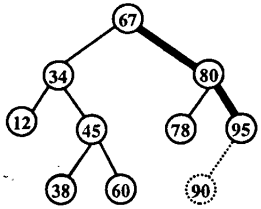
We insert the new key when we reach a NULL left or right child.



Insert 39
39 < 67, move to left child
39 > 34, move to right child
39 < 45, move to left child
39 > 38, move to right child
NULL right child
Insert as right child of 38



Insert 10
10 < 67, move to left child
10 < 34, move to left child
10 < 12, move to left child
NULL left child
Insert as left child of 12



Insert 90
90 > 67, move to right child
90 > 80, move to right child
90 < 95, move to left child
NULL left child
Insert as left child of 95

Figure 6.29 Insertion in a Binary Search Tree

To insert 39, first it is searched in the tree and the search terminates because we get NULL right subtree of 38, so it is inserted as the right child of node 38. Let us create a binary search tree from the given keys-

50, 30, 60, 38, 35, 55, 22, 59, 94, 13, 98

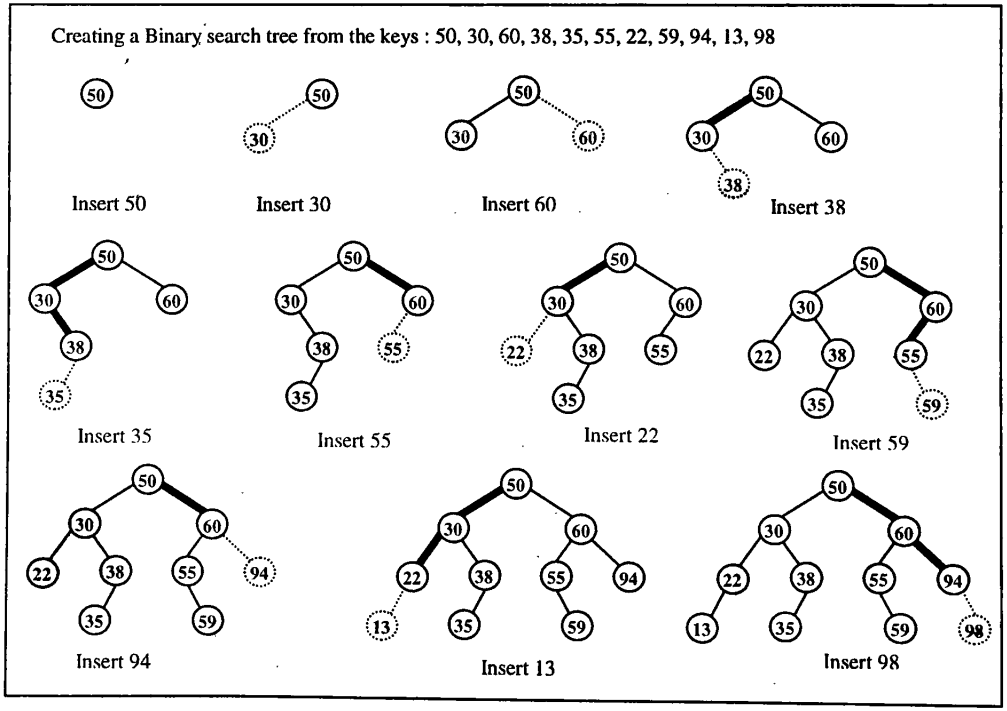


Figure 6.30

Note that if we alter the sequence of data we will get different binary search trees for the same data.

The non recursive function for the insertion of a node is given below. This function is passed the address of root of the tree and the key to be inserted. The root of the tree might change inside the function so it returns the address of root node.

```
struct node *insert_nrec(struct node *root,int ikey)
{
        struct node *tmp,*par,*ptr;
        ptr = root;
        par = NULL;
        while(ptr!=NULL)
        {
                par = ptr;
                if(ikey < ptr->info)
                        ptr = ptr->lchild;
                else if(ikey > ptr->info)
                        ptr = ptr->rchild;
                else
                {
                        printf("Duplicate key");
                        return root;
                }
        }
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = ikey;
        tmp->lchild = NULL;
        tmp->rchild = NULL;
        if(par==NULL)
                root = tmp;
        else if(ikey < par->info)
                par->lchild = tmp;
        else
                par->rchild = tmp;
        return root;
}/*End of insert_nrec()*/
```

This function will be called as `root = insert_nrec(root,ikey);`

To insert in empty tree, root is made to point to the new node. As we move down the tree, we keep track of the parent of the node because this is required for the insertion of new node. The pointer `ptr` walks down the path and the parent of the `ptr` is maintained through pointer `par`. When the search terminates unsuccessfully, `ptr` is NULL and `par` points to the node whose link should be changed to insert the new node. The new node `tmp` is made the left or right child of the parent `par`.

The insertion of a node in binary search tree can be described recursively also. First the key to be inserted is compared with key of root node. If the key is found there then there is nothing to be done so we return. If the key to be inserted is less than the key in the root node then it is inserted in left subtree otherwise it is inserted in right subtree. The same process is followed for both subtrees till we reach a situation where we have to insert node in an empty subtree.

The recursive function for the insertion of a node is given below. This function is passed the address of root of the tree and the key to be inserted. It returns the address of root of the tree. The two base cases which stop the recursion are – when we find the key or when we reach a NULL subtree.

```
struct node *insert(struct node *ptr,int ikey)
{
        if(ptr==NULL)    /*Base Case*/
        {
                ptr = (struct node *)malloc(sizeof(struct node));
                ptr->info = ikey;
                ptr->lchild = NULL;
                ptr->rchild = NULL;
        }
        else if(ikey < ptr->info)      /*Insertion in left subtree*/
                ptr->lchild = insert(ptr->lchild, ikey);
        else if(ikey > ptr->info)      /*Insertion in right subtree */
                ptr->rchild = insert(ptr->rchild, ikey);
        else
                printf("Duplicate key\n");   /*Base Case*/
```

```
        return ptr;
}/*End of insert()*/
```

Insertion operation takes O(h) time where h is the height of the tree.

6.12.5 Deletion in a Binary Search Tree

The node to be deleted is searched in the tree, if it is found then there can be three possibilities for that node-

A) Node has no child i.e. it is a leaf node.
B) Node has exactly 1 child.
C) Node has exactly 2 children.

The non recursive function for deletion of a node is given below. This function is passed the root of the tree and the key to be deleted. The root of the tree might be changed in the function so it returns the root of the tree. Inside the function, first the key is searched and if it is found then the three cases are handled in three different functions `case_a()`, `case_b()` and `case_c()`.

```
struct node *del_nrec(struct node *root,int dkey)
{
        struct node *par,*ptr;
        ptr = root;
        par = NULL;
        while(ptr!=NULL)
        {
                if(dkey==ptr->info)
                        break;
                par = ptr;
                if(dkey < ptr->info)
                        ptr = ptr->lchild;
                else
                        ptr = ptr->rchild;
        }
        if(ptr==NULL)
                printf("dkey not present in tree\n");
        else if(ptr->lchild != NULL && ptr->rchild != NULL)/*2 children*/
                root = case_c(root,par,ptr);
        else if(ptr->lchild != NULL )/*only left child*/
                root = case_b(root,par,ptr);
        else if(ptr->rchild != NULL)/*only right child*/
                root = case_b(root,par,ptr);
        else /*no child*/
                root = case_a(root,par,ptr);
        return root;
}/*End of del_nrec()*/
```

Case A :

To delete a leaf node N, the link to node N is replaced by NULL. If the node is left child of its parent then the left link of its parent is set to NULL and if the node is right child of its parent then the right link of its parent is set to NULL. Then the node is deallocated using `free()`.

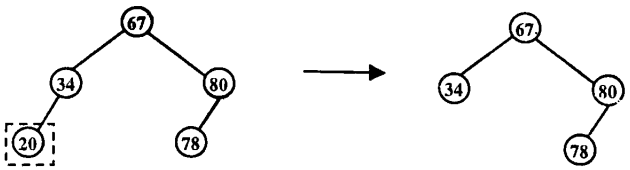


Figure 6.31 Deletion of 20

To delete 20, the left link of node 34 is set to NULL.

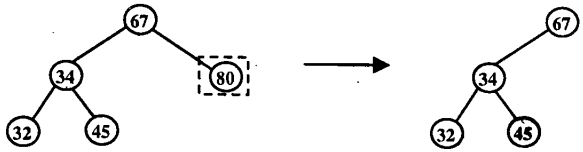


Figure 6.32 Deletion of 80

To delete 80, the right link of 67 is set to NULL.

If the leaf node to be deleted has no parent i.e. it is the root node then pointer `root` is set to NULL.

```
struct node *case_a(struct node *root,struct node *par,struct node *ptr)
{
        if(par==NULL) /*root node to be deleted*/
                root = NULL;
        else if(ptr==par->lchild)
                par->lchild = NULL;
        else
                par->rchild = NULL;
        free(ptr);
        return root;
}/*End of case_a()*/
```

Case B :

In this case, the node to be deleted has only one child. After deletion this single child takes the place of the deleted node. For this we just change the appropriate pointer of the parent node so that after deletion it points to the child of deleted node. After this, the node is deallocated using `free()`. Suppose N is the node to be deleted, P is its parent and C is its child.

If N is left child of P, then after deletion the node C becomes left child of P.
If N is right child of P, then after deletion the node C becomes right child of P.

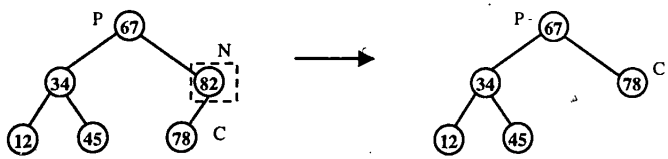


Figure 6.33 Deletion of 82

The node 82 is to be deleted from the tree. Node 82 is right child of its parent 67, so the single child 78 takes the place of 82 by becoming the right child of 67.

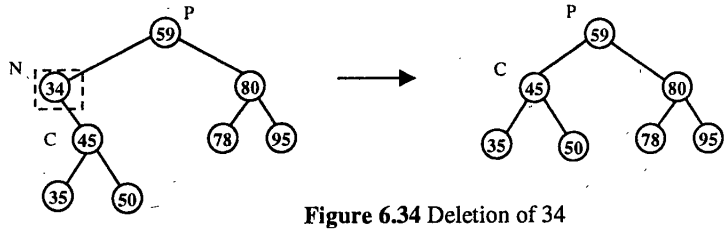


Figure 6.34 Deletion of 34

The node 34 is to be deleted from the tree. Node 34 is left child of its parent 59, so the single child 45 takes the place of 34 by becoming the left child of 59.

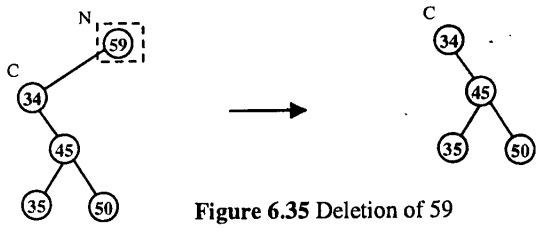


Figure 6.35 Deletion of 59

The node 59 is to be deleted from the tree. After searching we will find that its parent is NULL because it is the root node. After deletion the single child 34 will become the new root of the tree.

```
struct node *case_b(struct node *root,struct node *par,struct node *ptr)
{
        struct node *child;
        /*Initialize child*/
        if(ptr->lchild!=NULL) /*node to be deleted has left child */
                child = ptr->lchild;
        else                  /*node to be deleted has right child */
                child = ptr->rchild;
        if(par==NULL)    /*node to be deleted is root node*/
                root = child;
        else if(ptr==par->lchild)    /*node is left child of its parent*/
                par->lchild = child;
        else                         /*node is right child of its parent*/
                par->rchild = child;
        free(ptr);
        return root;
}/*End of case_b()*/
```

First we check whether the node to be deleted has left child or right child. If it has only left child then we store the address of left child and if it has only right child then we store the address of right child in pointer variable `child`.

If node to be deleted is root node then we assign `child` to `root`. So child of deleted node(root) will become the new root node. Otherwise, we check whether the node to be deleted is left child or right child of its parent. If it is left child then we assign `child` to `lchild` part of its parent otherwise we assign `child` to `rchild` part of its parent.

Case C :

This is the case when the node to be deleted has two children. Here we have to find the inorder successor of the node. The data of the inorder successor is copied to the node and then the inorder successor is deleted from the tree.

Inorder successor of a node can be deleted by case A or case B because it will have either one right child or no child, it can't have two children.

Inorder successor of a node is the node that comes just after that node in the inorder traversal of the tree. Inorder successor of a node N, having a right child is the leftmost node in the right subtree of the node N. To find the inorder successor of a node N we move to the right child of N and keep on moving left till we find a node with no left child. So we can see why inorder successor can't have a left child.

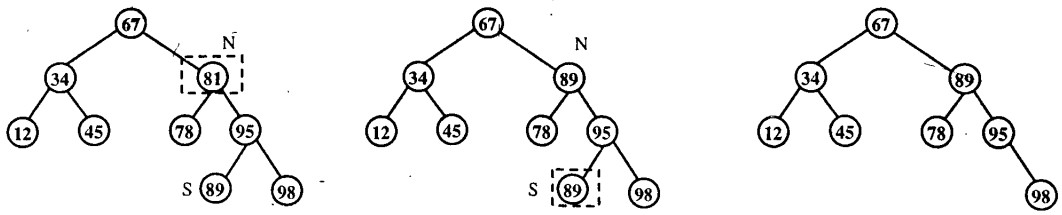


Figure 6.36 Deletion of 81

Here node N having the key 81 is to be deleted from the tree. Its inorder successor is node S having the key 89, so the data of node S is copied to node N and now node S is to be deleted from the tree. Node S can be deleted using case A because it has no child.

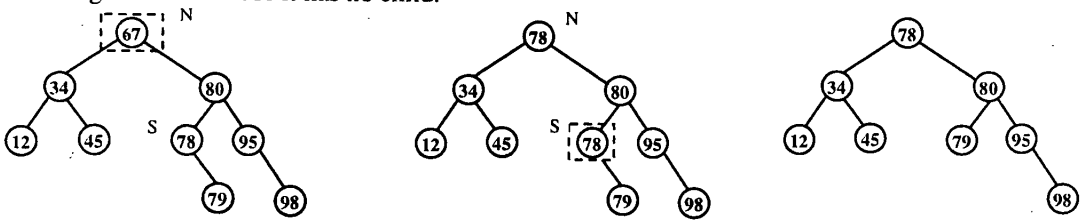


Figure 6.37 Deletion of 67

Here node N having the key 67 is to be deleted from the tree. Its inorder successor is node S having the key 78, so the data of node S is copied to node N and now node S has to be deleted from the tree. Node S can be deleted using case B because it has only one child.

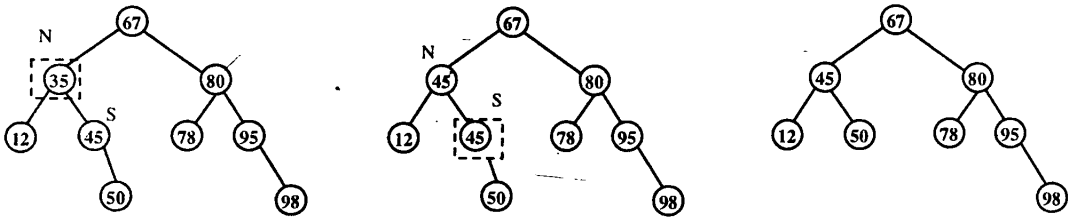


Figure 6.38 Deletion of 35

Here node N having the key 35 is to be deleted from the tree. Its inorder successor is node S having the key 45, so the data of node S is copied to node N and now node S has to be deleted from the tree. Node S can be deleted using case B because it has only one child.

To delete inorder successor we have to call `case_a()` or `case_b()` depending on whether it has no child or one child. For deleting a node by `case_a()` or `case_b()` we need to send the address of that node and address of its parent. So to delete inorder successor we need to find out its address and address of its parent. In the function `case_c()` we first find out the inorder successor and its parent, and then copy the information of successor node to the node to be deleted. After this the successor is deleted by calling `case_a()` or `case_b()`.

```
struct node *case_c(struct node *root, struct node *par,struct node *ptr)
{
        struct node *succ,*parsucc;
        /*Find inorder successor and its parent*/
        parsucc = ptr;
        succ = ptr->rchild;
        while(succ->lchild != NULL)
        {
                parsucc = succ;
                succ = succ->lchild;
        }
        ptr->info = succ->info;
        if(succ->lchild == NULL && succ->rchild == NULL)
                root = case_a(root,parsucc,succ);
        else
                root = case_b(root, parsucc,succ);
        return root;
}/*End of case_c()*/
```

In the following function `del_nrec1()`, we have not made separate functions for different cases. All the cases are handled in this function only. Case A and case B are regarded as one case, because in case A the `child` will be initialized to NULL.

```
struct node *del_nrec1(struct node *root,int dkey)
{
        struct node *par,*ptr,*child,*succ,*parsucc;
        ptr = root;
        par = NULL;
        while(ptr!=NULL)
        {
                if(dkey==ptr->info)
                        break;
                par = ptr;
                if(dkey < ptr->info)
                        ptr = ptr->lchild;
                else
                        ptr = ptr->rchild;
        }
        if(ptr==NULL)
        {
                printf("dkey not present in tree");
                return root;
        }
        /*Case C: 2 children*/
        if(ptr->lchild!=NULL && ptr->rchild!=NULL)
        {
                parsucc = ptr;
                succ = ptr->rchild;
                while(succ->lchild!=NULL)
                {
                        parsucc = succ;
                        succ = succ->lchild;
                }
                ptr->info = succ->info;
                ptr = succ;
                par = parsucc;
        }
        /*Case B and Case A : 1 or no child*/
        if(ptr->lchild!=NULL) /*node to be deleted has left child*/
                child = ptr->lchild;
        else                  /*node to be deleted has right child or no child*/
                child = ptr->rchild;
        if(par==NULL )   /*node to be deleted is root node*/
                root = child;
        else if(ptr==par->lchild)/*node is left child of its parent*/
                par->lchild = child;
        else       /*node is right child of its parent*/
                par->rchild = child;
        free(ptr);
        return root;
}
```

The recursive function for deletion from binary search tree is given below.

```
struct node *del(struct node *ptr,int dkey)
{
        struct node *tmp,*succ;
        if(ptr==NULL)
        {
                printf("dkey not found\n");
                return(ptr);
        }
        if(dkey < ptr->info)/*delete from left subtree*/
                ptr->lchild = del(ptr->lchild, dkey);
        else if(dkey > ptr->info)/*delete from right subtree*/
                ptr->rchild = del(ptr->rchild,dkey);
        else
        {
                /*key to be deleted is found*/
```

```
                if(ptr->lchild!=NULL && ptr->rchild!=NULL) /*2 children*/
                {
                        succ = ptr->rchild;
                        while(succ->lchild)
                                succ = succ->lchild;
                        ptr->info = succ->info;
                        ptr->rchild = del(ptr->rchild, succ->info);
                }
                else
                {
                        tmp = ptr;
                        if(ptr->lchild!=NULL) /*only left child*/
                                ptr = ptr->lchild;
                        else if(ptr->rchild!=NULL) /*only right child*/
                                ptr = ptr->rchild;
                        else    /*no child*/
                                ptr = NULL;
                        free(tmp);
                }
        }
        return ptr;
}/*End of del()*/
```

We can use the inorder predecessor instead of inorder successor in case C, and in that case we get a different binary search tree. Inorder predecessor of a node N is the rightmost node in the left subtree of node N. Like other operations, deletion also takes O(h) running time where h is the height of the tree.The `main()` function for all recursive operations is given next.

```
/*P6.3 Recursive operations in Binary Search Tree*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        struct node *lchild;
        int info;
        struct node *rchild;
};
struct node *search(struct node *ptr,int skey);
struct node *insert(struct node *ptr,int ikey);
struct node *del(struct node *ptr,int dkey);
struct node *Min(struct node *ptr);
struct node *Max(struct node *ptr);
void preorder(struct node *ptr);
void inorder(struct node *ptr);
void postorder(struct node *ptr);
int height(struct node *ptr);

main()
{
        struct node *root=NULL,*ptr;
        int choice,k;
        while(1)
        {
                printf("\n");
                printf("1.Search\n");
                printf("2.Insert\n");
                printf("3.Delete\n");
                printf("4.Preorder Traversal\n");
                printf("5.Inorder Traversal\n");
                printf("6.Postorder Traversal\n");
                printf("7.Height of tree\n");
                printf("8.Find minimum and maximum\n");
                printf("9.Quit\n");
                printf("Enter your choice : ");
```

```
            scanf("%d",&choice);
            switch(choice)
            {
            case 1:
                    printf("Enter the key to be searched : ");
                    scanf("%d",&k);
                    ptr = search(root,k);
                    if(ptr==NULL)
                            printf("Key not present\n");
                    else
                            printf("Key present\n");
                    break;
            case 2:
                    printf("Enter the key to be inserted : ");
                    scanf("%d",&k);
                    root = insert(root,k);
                    break;
            case 3:
                    printf("Enter the key to be deleted : ");
                    scanf("%d",&k);
                    root = del(root,k);
                    break;
             case 4:
                    preorder(root);
                    break;
             case 5:
                    inorder(root);
                    break;
             case 6:
                    postorder(root);
                    break;
             case 7:
                     printf("Height of tree is %d\n", height(root));
                     break;
             case 8:
                    ptr = Min(root);
                    if(ptr!=NULL)
                            printf("Minimum key is %d\n",ptr->info);
                    ptr = Max(root);
                    if(ptr!=NULL)
                            printf("Maximum key is %d\n",ptr->info);
                    break;
             case 9:
                    exit();
             default:
                    printf("Wrong choice\n");
           }/*End of switch*/
      }/*End of while*/
}/*End of main()*/
```

The functions for traversals and height are same as given in binary tree.

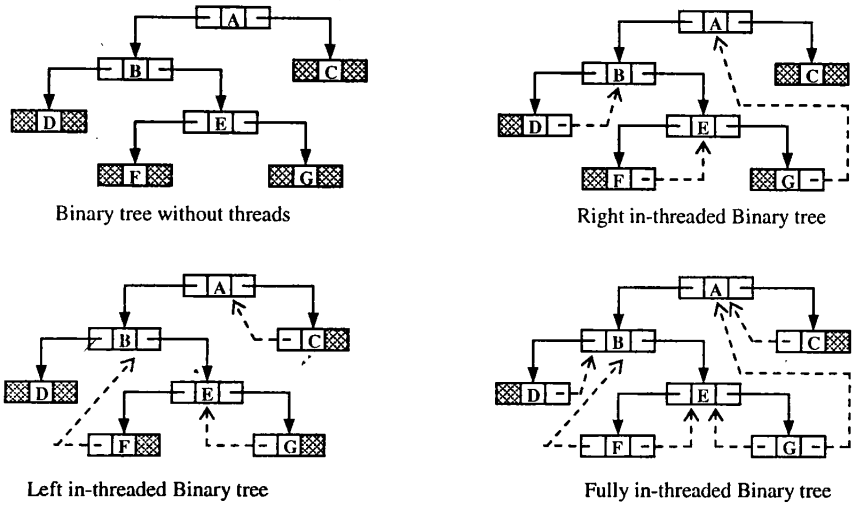
6.13 Threaded Binary Tree

We had written the traversals of binary tree in two ways – recursive and non recursive. Both types of algorithms required the use of stack; recursive version used a run-time stack and non recursive version used a user defined stack. Traversal is a common operation and if the tree has to be traversed frequently then using a stack is not efficient since extra time and space is needed in maintaining the stack. The traversals can be implemented more efficiently if we can avoid the use of stack. We can do this by maintaining threads in the binary tree.

A binary tree with n nodes has 2n pointers out of which n+1 are always NULL(Property 11), so we can see that about half the space allocated for pointers is wasted. We can utilize this wasted space to contain some useful information. A left NULL pointer can be used to store the address of inorder predecessor of the node and

a right NULL pointer can be used to store the address of inorder successor of the node. These pointers are called threads and a binary tree which implements these pointers is called a threaded binary tree.

If only left NULL pointers are used as threads then the binary tree is called a left in-threaded binary tree, if only right NULL pointers are used as threads then the binary tree is called a right in-threaded binary tree, and if both left and right NULL pointers are used as threads then the binary tree is called fully in-threaded tree or in-threaded tree. The following figure shows these threading in a binary tree.



Binary tree without threads

Right in-threaded Binary tree

Left in-threaded Binary tree

Fully in-threaded Binary tree

Figure 6.39

The inorder traversal of the tree in figure 6.39 is [D B F E G A C].

D's right link is NULL and its successor is B, so right pointer of D is made a thread.
F's right link is NULL and its successor is E, so right pointer of F is made a thread.
G's right link is NULL and its successor is A, so right pointer of G is made a thread.
F's left link is NULL and its predecessor is B, so left pointer of F is made a thread.
G's left link is NULL and its predecessor is E, so left pointer of G is made a thread.
C's left link is NULL and its predecessor is A, so left pointer of C is made a thread.

D's left link is NULL but it has no predecessor and C's right link is NULL but it has no successor so both these pointer fields are NULL in the fully threaded tree.

Similar threading can be done corresponding to the other two traversals also i.e. postorder and preorder. In our discussion we'll take inorder threading only.

While implementing a threaded binary tree we should be able to distinguish real children pointers from threads. For this we can attach two boolean variables to each node, and these variables will be used to determine whether left and right pointers of that node are thread or child pointer. The structure declaration of a node in a fully in-threaded binary tree will be as-

```
struct node
{
        struct node *left;
        boolean lthread;
        int info;
        boolean rthread;
        struct node *right;
};
```

Here we have taken two boolean members `lthread` and `rthread` to differentiate between a thread and child pointer. If `lthread` is true then left pointer is a thread to inorder predecessor otherwise it contains address of left child, similarly if `rthread` is true then right pointer is a thread to inorder successor otherwise it contains address of right child.

First node in inorder traversal has no predecessor and last node has no successor. So the left pointer of the first node and the right pointer of the last node in inorder traversal contain NULL. For consistency the `lthread` field of first node and `rthread` field of last node are set to true.All the declarations and `main()` function for the program of threaded tree are given below.

```
/*P6.5 Insertion, Deletion and Traversal in fully in-threaded Binary Search Tree*/
#include<stdio.h>
#include<stdlib.h>
typedef enum {false,true} boolean;
struct node *in_succ(struct node *p);
struct node *in_pred(struct node *p);
struct node *insert(struct node *root,int ikey);
struct node *del(struct node *root,int dkey);
struct node *case_a(struct node *root,struct node *par,struct node *ptr);
struct node *case_b(struct node *root,struct node *par,struct node *ptr);
struct node *case_c(struct node *root,struct node *par,struct node *ptr);
void inorder(struct node *root);
void preorder(struct node *root);
struct node
{
        struct node *left;
        boolean lthread;
        int info;
        boolean rthread;
        struct node *right;
};

main()
{
        int choice,num;
        struct node *root=NULL;
        while(1)
        {
                printf("\n");
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Inorder Traversal\n");
                printf("4.Preorder Traversal\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1:
                        printf("Enter the number to be inserted : ");
                        scanf("%d",&num);
                        root = insert(root,num);
                        break;
                 case 2:
                        printf("Enter the number to be deleted : ");
                        scanf("%d",&num);
                        root = del(root,num);
                        break;
                 case 3:
                        inorder(root);
                        break;
                 case 4:
                        preorder(root);
                        break;
                 case 5:
                         exit(1);
                 default:
                        printf("Wrong choice\n");
                }/*End of switch*/
```

```
        }/*End of while*/
}/*End of main()*/
```

6.13.1 Finding inorder successor of a node in in-threaded tree

If the right pointer of the node is a thread, then there is no need to do anything because the right thread points to inorder successor. If node's right pointer is not a thread i.e. node has a right child then to find the inorder successor, we move to this right child and keep on moving left till we find a node with no left child.

```
struct node *in_succ(struct node *ptr)
{
        if(ptr->rthread==true)
                return ptr->right;
        else
        {
                ptr = ptr->right;
                while(ptr->lthread==false)
                        ptr = ptr->left;
                return ptr;
        }
}/*End of in_succ()*/
```

6.13.2 Finding inorder predecessor of a node in in-threaded tree

If the left pointer is a thread, then this thread will point to the inorder predecessor. If the left pointer is not a thread i.e. node has a left child then to find the inorder predecessor, we move to this left child and keep on moving right till we find a node with no right child.

```
struct node *in_pred(struct node *ptr)
{
        if(ptr->lthread==true)
                return ptr->left;
        else
        {
                ptr = ptr->left;
                while(ptr->rthread==false)
                        ptr = ptr->right;
                return ptr;
        }
}/*End of in_pred()*/
```

6.13.3 Inorder Traversal of in-threaded binary tree

The first node to be visited in the inorder traversal is the last node in the leftmost branch starting from root. So we start from the root and move left till we get a node with no left child, and this node is visited. Now with the help of `in_succ()` function we find the inorder successor of each node and visit it. The right pointer of last node is NULL and we have marked it as thread. So we stop our process when we get a node whose right thread is NULL.

```
inorder(struct node *root)
{
        struct node *ptr;
        if(root==NULL )
        {
                printf("Tree is empty");
                return;
        }
        ptr = root;
        /*Find the leftmost node*/
        while(ptr->lthread==false)
                ptr = ptr->left;
```

```
        while(ptr!=NULL)
        {
                printf("%d ",ptr->info);
                ptr = in_succ(ptr);
        }
}/*End of inorder()*/
```

6.13.4 Preorder traversal of in-threaded binary tree

In preorder traversal first the root is visited, so we start from the root node. If the node has a left child then that left child will be visited, otherwise if the node has a right child then that right child will be visited. If the node has neither left child nor right child (it is a leaf node) then with the help of right threads we will reach that inorder successor of the node which has a right subtree. Now this subtree will be traversed in preorder in the same way.

```
preorder(struct node *root)
{
        struct node *ptr;
        if(root==NULL)
        {
                printf("Tree is empty");
                return;
        }
        ptr = root;
        while(ptr!=NULL)
        {
                printf("%d ",ptr->info);
                if(ptr->lthread==false)
                        ptr = ptr->left;
                else if(ptr->rthread==false)
                        ptr = ptr->right;
                else
                {
                        while(ptr!=NULL && ptr->rthread==true)
                                ptr = ptr->right;
                        if(ptr!=NULL)
                                ptr = ptr->right;
                }
        }
}/*End of preorder()*/
```

6.13.5 Insertion and deletion in threaded binary tree

Insertion and deletion in threaded binary tree will be same as insertion and deletion in a binary tree but we will have to adjust the threads after these operations. We'll take the example of threaded binary search tree and see how the nodes can be inserted and deleted from it.

6.13.5.1 Insertion

As in binary search tree here also first we will search for the key value in the tree, if it is already present then we return otherwise the new key is inserted at the point where search terminates. In BST, search terminates either when we find the key or when we reach a NULL left or right pointer. Here all left and right NULL pointers are replaced by threads except left pointer of first node and right pointer of last node. So here search will be unsuccessful when we reach a NULL pointer or a thread.

The new node tmp will be inserted as a leaf node so its left and right pointers both will be threads.

```
tmp = (struct node *)malloc(sizeof(struct node));
tmp->info = ikey;
tmp->lthread = true;
tmp->rthread = true;
```

Case 1 : Insertion in empty tree

Both left and right pointers of tmp will be set to NULL and new node becomes the root.

```
root = tmp;
tmp->left = NULL;
tmp->right = NULL;
```

Case 2- When new node inserted as the left child

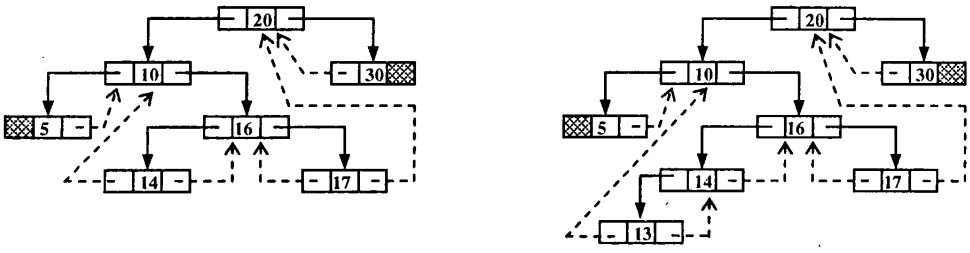
After inserting the node at its proper place we have to make its left and right threads point to inorder predecessor and successor respectively. The node which was inorder predecessor of the parent is now the inorder predecessor of this node tmp. The parent of tmp is its inorder successor. So the left and right threads of the new node will be-

```
tmp->left = par->left;
tmp->right = par;
```

Before insertion, the left pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

```
par->lthread = false;
par->left = tmp;
```

The following example shows a node being inserted as left child of its parent.



Insert 13
Inorder : 5 10 14 16 17 20 30

13 inserted as left child of 14
Inorder : 5 10 13 14 16 17 20 30

Figure 6.40 Insertion of 13

Predecessor of 14 becomes the predecessor of 13, so left thread of 13 points to 10
Successor of 13 is 14, so right thread of 13 points to 14
Left pointer of 14 is not a thread now, it points to left child which is 13

Case 3 - When new node is inserted as the right child.

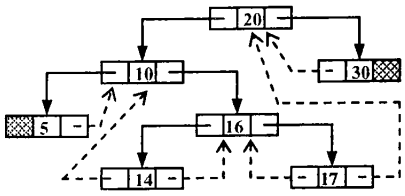
The parent of tmp is its inorder predecessor. The node which was inorder successor of the parent is now the inorder successor of this node tmp. So the left and right threads of the new node will be-

```
tmp->left = par;
tmp->right = par->right;
```

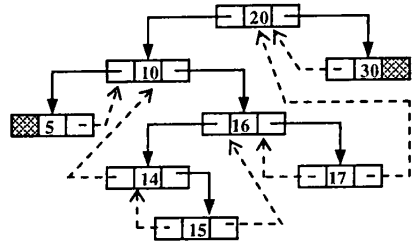
Before insertion, the right pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

```
par->rthread = false;
par->right = tmp;
```

The following example shows a node being inserted as right child of its parent.



Insert 15
Inorder : 5 10 14 16 17 20 30



15 inserted as right child of 14
Inorder : 5 10 14 15 16 17 20 30

Figure 6.41 Insertion of 15

Successor of 14 becomes the successor of 15, so right thread of 15 points to 16
Predecessor of 15 is 14, so left thread of 15 points to 14.
Right pointer of 14 is not a thread now, it points to right child which is 15

```
struct node *insert(struct node *root,int ikey)
{
        struct node *tmp,*par,*ptr;
        int found = 0;
        ptr = root;     par = NULL;
        while(ptr!=NULL)
        {
                if(ikey == ptr->info)
                {
                        found = 1;
                        break;
                }
                par = ptr;
                if(ikey < ptr->info)
                {
                        if(ptr->lthread == false)
                                ptr = ptr->left;
                        else
                                break;
                }
                else
                {
                        if(ptr->rthread == false)
                                ptr = ptr->right;
                        else
                                break;
                }
        }
        if(found)
                printf("Duplicate key");
        else
        {
                tmp = (struct node *)malloc(sizeof(struct node));
                tmp->info = ikey;
                tmp->lthread = true;
                tmp->rthread = true;
                if(par==NULL)
                {
                        root = tmp;
                        tmp->left = NULL;
                        tmp->right = NULL;
                }
                else if(ikey < par->info)
                {
                        tmp->left = par->left;
```

```
                        tmp->right = par;
                        par->lthread = false;
                        par->left = tmp;
                }
                else
                {
                        tmp->left = par;
                        tmp->right = par->right;
                        par->rthread = false;
                        par->right = tmp;
                }
        }
        return root;
}/*End of insert()*/
```

6.13.5.2 Deletion

First the key to be deleted is searched, and then there are different cases for deleting the node in which key is found.

```
struct node *del(struct node *root,int dkey)
{
        struct node *par,*ptr;
        int found = 0;
        ptr = root;  par = NULL;
        while(ptr!=NULL)
        {
                if(dkey==ptr->info)
                {
                        found = 1;
                        break;
                }
                par = ptr;
                if(dkey < ptr->info)
                {
                        if(ptr->lthread==false)
                                ptr = ptr->left;
                        else
                                break;
                }
                else
                {
                        if(ptr->rthread==false)
                                ptr = ptr->right;
                        else
                                break;
                }
        }
        if(found==0)
                printf("dkey not present in tree");
        else if(ptr->lthread==false && ptr->rthread==false)/*2 children*/
                root = case_c(root,par,ptr);
        else if(ptr->lthread==false)/*only left child*/
                root = case_b(root, par,ptr);
        else if(ptr->rthread==false)/*only right child*/
                root = case_b(root, par,ptr);
        else /*no child*/
                root = case_a(root,par,ptr);
        return root;
/*End of del()*/
```

Case A : Leaf node to be deleted

In BST, for deleting a leaf node the left or right pointer of parent was set to NULL. Here instead of setting the pointer to NULL it is made a thread.

If the leaf node to be deleted is left child of its parent then after deletion, left pointer of parent should become a thread pointing to its predecessor. The node which was inorder predecessor of the leaf node before deletion, will become the inorder predecessor of the parent node after deletion.

```
par->lthread = true;    par->left = ptr->left;
```

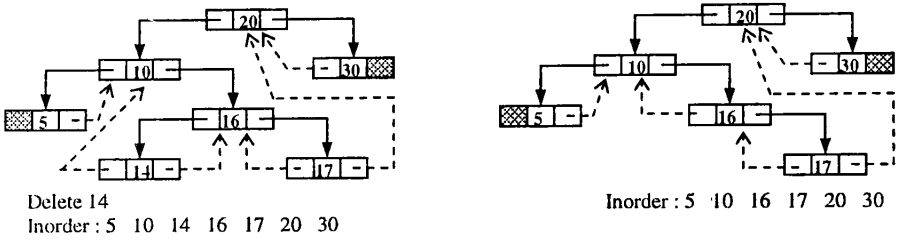


Figure 6.42 Deletion of 14

If the leaf node to be deleted is right child of its parent then after deletion, right pointer of parent should become a thread pointing to its successor. The node which was inorder successor of the leaf node before deletion will become the inorder successor of the parent node after deletion.

```
par->rthread = true;    par->right = ptr->right;
```

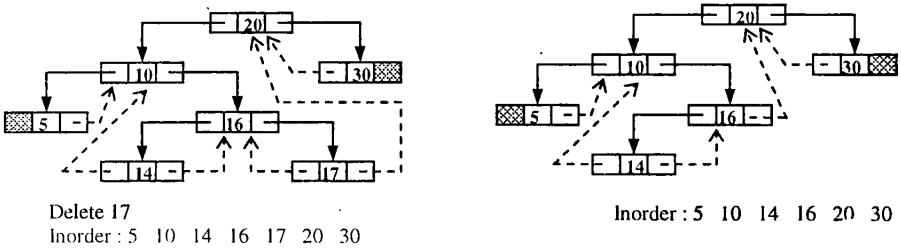


Figure 6.43 Deletion of 17

```
struct node *case_a(struct node *root, struct node *par,struct node *ptr )
{
        if(par == NULL) /*key to be deleted is in root node*/
                root = NULL;
        else if(ptr == par->left)
        {
                par->lthread = true;
                par->left = ptr->left;
        }
        else
        {
                par->rthread = true;
                par->right = ptr->right;
        }
        free(ptr);
        return root;
}/*End of case_a()*/
```

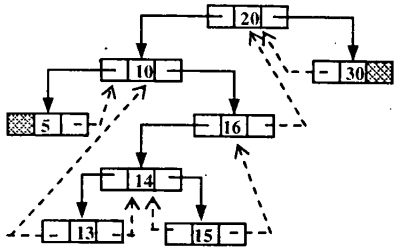
Case B : Node to be deleted has only one child

After deleting the node as in a BST, the inorder successor and inorder predecessor of the node are found out.

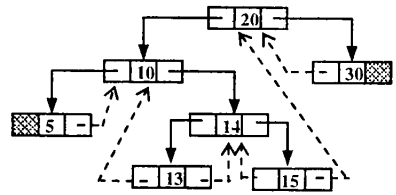
```
s = in_succ(ptr);
p = in_pred(ptr);
```

If node to be deleted has left subtree, then after deletion right thread of its predecessor should point to its successor.

```
p->right = s;
```



Delete 16
Inorder : 5 10 13 14 15 16 20 30



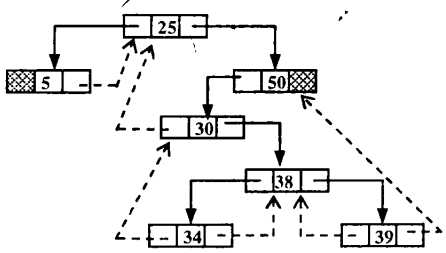
Inorder : 5 10 13 14 15 20 30

Figure 6.44 Deletion of 16

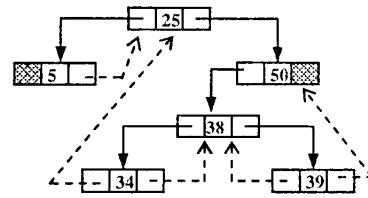
Before deletion, 15 is predecessor and 20 is successor of 16. After deletion of 16, the node 20 becomes the successor of 15, so right thread of 15 will point to 20.

If node to be deleted has right subtree, then after deletion left thread of its successor should point to its predecessor.

```
s->left = p;
```



Delete 30
Inorder : 5 25 30 34 38 39 50



Inorder : 5 25 34 38 39 50

Figure 6.45 Deletion of 30

Before deletion 25 is predecessor and 34 is successor of 30. After deletion of 30, the node 25 becomes the predecessor of 34, so left thread of 34 will point to 25.

```
struct node *case_b(struct node *root,struct node *par,struct node *ptr)
{
        struct node *child,*s,*p;
        /*Initialize child*/
        if(ptr->lthread==false) /*node to be deleted has left child*/
                child = ptr->left;
        else                    /*node to be deleted has right child*/
                child = ptr->right;
        if(par==NULL)       /*node to be deleted is root node*/
                root = child;
        else if(ptr==par->left)    /*node is left child of its parent*/
                par->left = child;
        else                    /*node is right child of its parent*/
                par->right = child;
        s = in_succ(ptr);
        p = in_pred(ptr);
        if(ptr->lthread==false)                 /*if ptr has left subtree*/
                p->right = s;
        else
        {
                if(ptr->rthread == false)  /*if ptr has right subtree*/
                        s->left = p;
        }
        free(ptr);
        return root;
```

```
}/*End of case_b()*/
```

Case C : Node to be deleted has two children

Suppose node N is to be deleted, first we will find the inorder successor of node N and then copy the information of this inorder successor into node N. After this inorder successor node is deleted using either case_a or case_b.

```
struct node *case_c(struct node *root, struct node *par,struct node *ptr)
{
        struct node *succ,*parsucc;
        /*Find inorder successor and its parent*/
        parsucc = ptr;
        succ = ptr->right;
        while(succ->left != NULL)
        {
                parsucc = succ;
                succ = succ->left;
        }
        ptr->info = succ->info;
        if(succ->lthread == true && succ->rthread == true)
                root = case_a(root, parsucc,succ);
        else
                root = case_b(root, parsucc,succ);
        return root;
}/*End of case_c()*/
```

6.13.6 Threaded tree with Header node

In the inorder traversal, the first node has no predecessor and last node has no successor. So in an in-threaded binary tree, the left pointer of first node and right pointer of last node are NULL. To make the threading consistent, we can take a dummy node called the header node which can act as predecessor of the first node and the successor of last node. Now the left pointer of first node and right pointer of last node will be threads pointing to this header node.

The binary tree can be represented as the left subtree of this header node. Left pointer of this header node will point to the root node of our tree and right pointer will point to itself. If we take a header node in the in-threaded tree of figure 6.39, the tree will look like this.

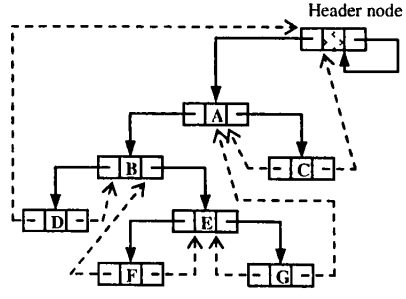
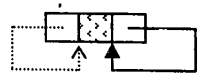


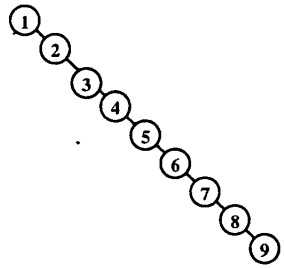
Figure 6.46 Threaded Tree with header node

In an empty tree, the left pointer of header node will be a thread pointing to itself.

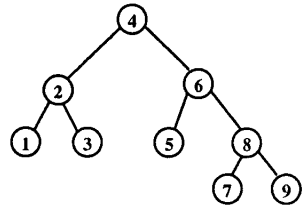


6.14 AVL Tree

A set of data can produce different binary search trees if the sequence of insertion of elements is different. Suppose we have data that consists of numbers from 1 to 9. If we insert the data in sequence {1, 2, 3, 4, 5, 6, 7, 8, 9}, then we get a binary search tree shown in figure 6.47(a) while if we insert the data in sequence {4, 2, 6, 1, 3, 5, 8, 7, 9}, we get a binary search tree shown in figure 6.47 (b). Let us compare the search efficiency of these two trees. For this, we will find out the average number of comparisons that have to be done to reach a node.



(a)Average number of comparisons to reach a node
=(1+2+3+4+5+6+7+8+9)/ 9 = 45/9 = 5



(b)Average number of comparisons to reach a node
= (1+2+2+3+3+3+3+4+4+) = 25/9 = 2.77

Figure 6.47

In tree (a), we need one comparison to reach node 1, two comparisons to reach node 2, nine comparisons to reach node 9. Therefore, the average number of comparisons to reach a node is 5. In tree (b), we need one comparison to reach node 4, two comparisons each to reach node 2 or 6, three comparisons each to reach node 1, 3, 5 or 8, four comparisons each to reach node 7 or 9. In this tree, average number of comparisons is 2.77.

We can see that the tree (b) is better than tree (a) from searching point of view. In fact tree (a) is a form of linear list and searching is O(n). Both the trees have same data but their structures are different because of different sequence of insertion of elements. It is not possible to control the order of insertion so the concept of height balanced binary search trees came in. The technique for balancing a binary search tree was introduced by Russian mathematicians G. M. Adelson-Velskii and E. M. Landis in 1962. The height balanced binary search tree is called AVL tree in their honour.

The main aim of AVL tree is to perform efficient search, insertion and deletion operations. Searching is efficient when the heights of left and right subtrees of the nodes are almost same. This is possible in a full or complete binary search tree, which is an ideal situation and is not always achievable. This ideal situation is very nearly approximated by AVL trees.

An AVL tree is a binary search tree where the difference in the height of left and right subtrees of any node can be at most 1. Let us take a binary search tree and find out whether it is AVL tree or not.

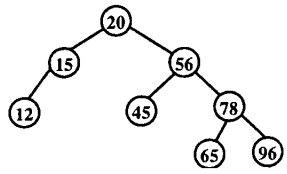


Figure 6.48

For the leaf nodes 12, 45, 65 and 96, left and right subtrees are empty so difference of heights of their subtrees is 0.

For node 20, height of left subtree is 2 and height of right subtree is 3, so difference is 1.
For node 15, height of left subtree is 1 and height of right subtree is 0, so difference is 1.
For node 56, height of left subtree is 1 and height of right subtree is 2, so difference is 1.
For node 78, height of left subtree is 1 and height of right subtree is 1, so difference is 0.

In this binary search tree, the difference in the height of left and right subtrees of any node is at most 1 and so it is an AVL tree. Now let us take another binary search tree.

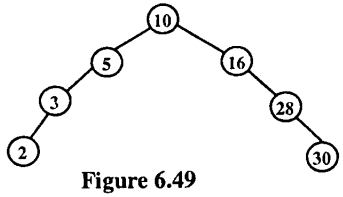


Figure 6.49

For leaf nodes 2 and 30, left and right subtrees are empty so difference is 0.
For node 10, height of left subtree is 3 and height of right subtree is 3, so difference is 0.
For node 5, height of left subtree is 2 and height of right subtree is 0, so difference is 2.
For node 3, height of left subtree is 1 and height of right subtree is 0, so difference is 1.
For node 16, height of left subtree is 0 and height of right subtree is 2, so difference is 2.
For node 28, height of left subtree is 0 and height of right subtree is 1, so difference is 1.

This tree is not an AVL tree since there are two nodes for which difference in heights of left and right subtrees exceeds 1.

Each node of an AVL tree has a balance factor, which is defined as the difference between the heights of left subtree and right subtree of a node.

Balance factor of a node = Height of its left subtree - Height of its right subtree

From the definition of AVL tree, it is obvious that the only possible values for the balance factor of any node are -1, 0, 1.

A node is called right heavy or right high if height of its right subtree is one more than height of its left subtree. A node is called left heavy or left high if height of its left subtree is one more than height of its right subtree. A node is called balanced if the heights of its right and left subtrees are same. The balance factor is 1 for left high, -1 for right high and 0 for balanced node.

So while writing the program for AVL tree, we will take an extra member in the structure of the tree node, which will store the balance factor of the node.

```
struct node
{
        struct node *lchild;
        int info;
        struct node *rchild;
        int balance;
};
```

As in binary search tree, we will take an integer value in the info part of the node, which will serve as the key.

The three trees in figure 6.50 are examples of AVL trees. The balance factor of each node is shown outside the node. In all these trees, the balance factor of each node is -1, 0 or 1.

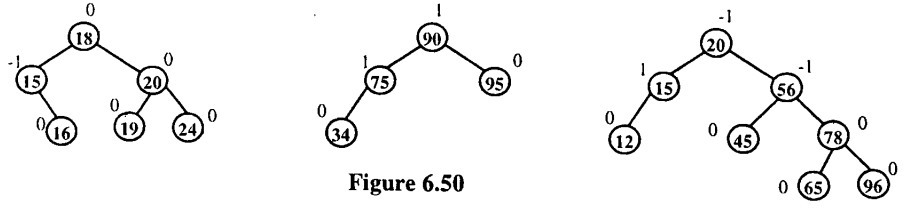


Figure 6.50

Let us see some trees that are binary search trees but not AVL trees.

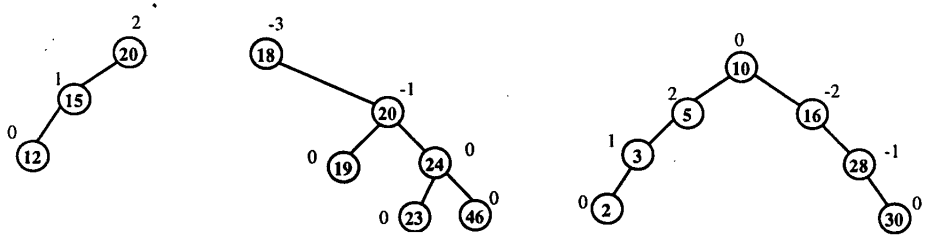


Figure 6.51

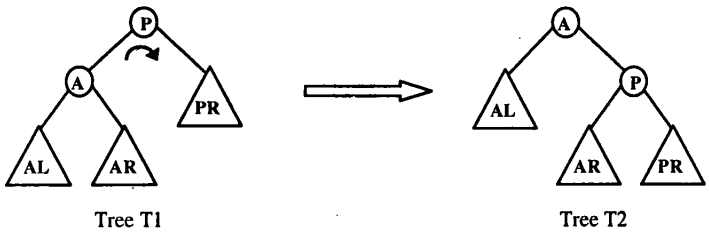
6.14.1 Searching and Traversal in AVL tree

Searching and traversal in AVL tree is done in the same way as in Binary search tree.

6.14.2 Tree Rotations

After insertion or deletion operations the balance factor of the nodes are affected and the tree might become unbalanced. The balance of the tree is restored by performing rotations, so before studying the insertion and deletion operations in AVL tree, we need to know about tree rotations. Rotations are simple manipulation of pointers which convert the tree in such a way that the new converted tree retains the binary search tree property with inorder traversal same as that of the original tree. Now let us see how right rotation and left rotation are performed in binary search trees.

6.14.2.1 Right Rotation



Tree T1 Tree T2

Figure 6.52 Right Rotation

In the tree T1, A is left child of node P and subtrees AL, AR are subtrees of node A. The subtree PR is right subtree of node P. Since T1 is a binary search tree, we can write-

keys(AL) < key(A) < keys(AR) < key(P) < keys (PR) .....(1)

The inorder traversal of tree T1 will be-

inorder(AL), A, inorder(AR), P, inorder (PR)

Now we perform a right rotation about the node P and the tree T1 will be transformed to tree T2 as shown in figure 6.52. The changes that take place are -

(i) Right subtree of A becomes left subtree of P

(ii) P becomes right child of A

(iii) A becomes the new root of the tree

If tree T2 has to satisfy the property of a binary search tree then the relationship among the keys should be-

keys(AL) < key(A) < keys(AR) < key(P) < keys (PR)

We have seen that this relation is true(from (1)), so tree T2 is also a binary search tree. The inorder traversal of tree T2 will be -

inorder(AL), A, inorder(AR), P, inorder (PR)

This is the same as the inorder traversal of tree T1.

Here T1 could be a binary search tree or a subtree of any node of a binary search tree. Let us illustrate this rotation with an example.

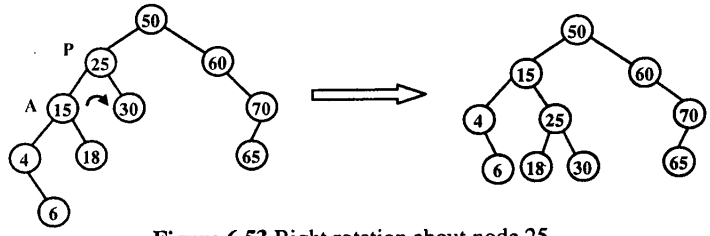


Figure 6.53 Right rotation about node 25

Here we are performing right rotation about node 25. If we compare this figure with the figure 6.52, then node 25 is the node P and node 15 is the node A. The subtree AL consists of nodes 4 and 6 while subtree AR consists of only node 18. The subtree PR consists of only node 30. The changes that take place due to right rotation are-

(i) Right subtree of node 15 becomes left subtree of node 25.

(ii) Node 25 becomes the right child of node 15.

(iii) The subtree which was rooted at node 25 is now rooted at node 15. So previously, the root of left subtree of node 50 was node 25, while after rotation the root of left subtree of node 50 is node 15.

Let us see two more examples of right rotation.

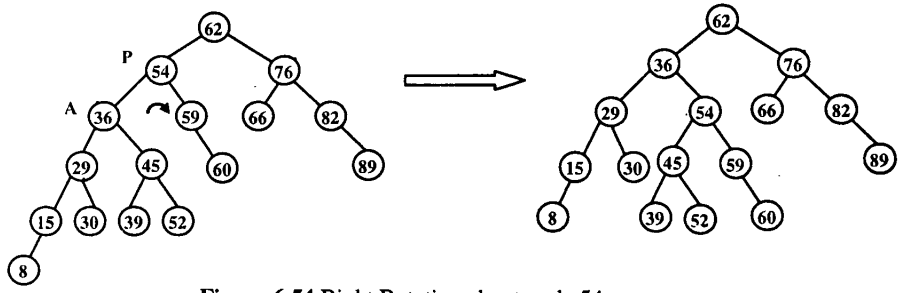


Figure 6.54 Right Rotation about node 54

Here right rotation is performed about node 54. The subtree PR consists of nodes 59 and 60, subtree AL consists of nodes 29, 15, 30, 8 and subtree AR consists of nodes 45, 39, 52.

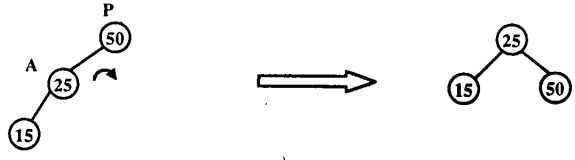


Figure 6.55 Right Rotation about node 50

Here right rotation is performed about node 50. The subtrees PR and AR are empty while subtree AL consists of node 15.

The function for right rotation is simple; it takes a pointer to node P, performs rotation and returns pointer to node A which is the new root of the subtree initially rooted at P.

```
struct node *RotateRight(struct node *pptr)
{
        struct node *aptr;
        aptr = pptr->lchild;      /*A is left child of P */
        pptr->lchild = aptr->rchild; /*Right child of A becomes left child of P*/
        aptr->rchild = pptr;      /*P becomes right child of A*/
        return aptr; /*A is the new root of the subtree initially rooted at P*/
}/*End of RotateRight()*/
```

6.14.2.2 Left Rotation

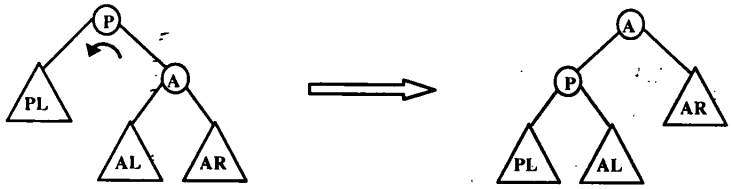


Figure 6.56 Left Rotation

In the tree T1, A is right child of node P and subtrees AL, AR are subtrees of node A. The subtree PL is left subtree of node P. Since T1 is a binary search tree, we can write-

keys(PL) < key(P) < keys(AL) < key(A) < keys(AR) .....(2)

The inorder traversal of this tree would be-

inorder(PL), P, inorder(AL), A, inorder(AR)

Now we perform a left rotation about the node P and the tree T1 will be transformed to tree T2. The changes that take place are -

(i) Left subtree of A becomes right subtree of P.
(ii) P becomes left child of A.
(iii) A becomes the new root of the tree.

If tree T2 has to satisfy the property of a binary search tree then the relationship among the keys should be-

keys(PL) < key(P) < keys(AL) < key(A) < keys(AR)

We have seen that this relation is true(from (2)), so tree T2 is also a binary search tree. The inorder traversal of binary search tree T2 will be-

inorder(PL), P, inorder(AL), A, inorder(AR)

This is same as the inorder traversal of the tree T1. Here is an example of left rotation.

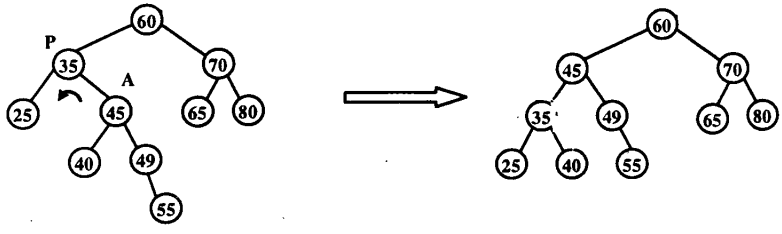


Figure 6.57 Left Rotation about node 35

Here we are performing left rotation about node 35. If we compare it with the figure 6.56, then node 35 is node P, node 45 is node A. The subtree AL consists of only node 40 and subtree AR consists of nodes 49 and 55. The subtree PL consists of only node 25. The changes that take place due to rotation are-

(i) Left subtree of node 45 becomes right subtree of node 35,
(ii) Node 35 becomes the left child of node 45
(iii) The subtree which was rooted at node 35 is now rooted at node 45.

Let us see two more examples of left rotation.

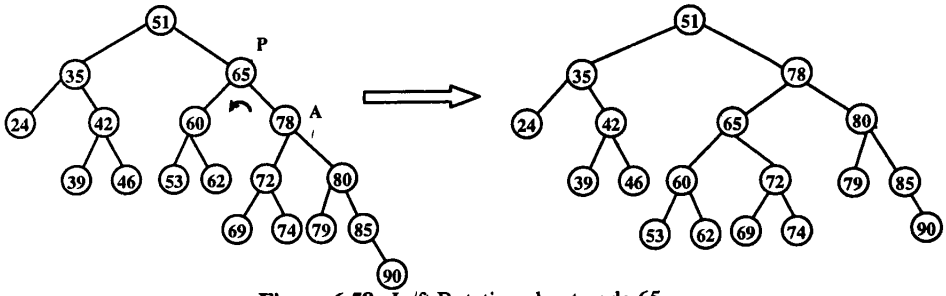


Figure 6.58 Left Rotation about node 65

Here we are performing left rotation about node 65. The subtree PL consists of nodes 60, 53 and 62, subtree AL consists of nodes 72, 69, 74, and subtree AR consists of nodes 80,79,85,90.

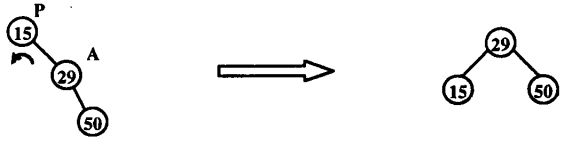


Figure 6.59 Left rotation about node 15

Here we are performing left rotation about node 15. The subtrees PL and AL are empty while subtree AR consists of node 50. The function for left rotation is-

```
struct node *RotateLeft(struct node *pptr)
{
        struct node *aptr;
        aptr = pptr->rchild;       /*A is right child of P*/
        pptr->rchild = aptr->lchild; /*Left child of A becomes right child of P*/
        aptr->lchild = pptr;      /*P becomes left child of A*/
        return aptr;  /*A is the new root of the subtree initially rooted at P*/
}/*End of RotateLeft()*/
```

Now we will see how to perform insertion and deletion operations in AVL tree. The `main()` function and other function declarations required for the AVL tree program are given next. The definitions of the functions are given with the explanation of the procedures.

```
/*P6.6 Program of AVL tree*/
#include<stdio.h>
#include<stdlib.h>
#define FALSE 0
#define TRUE 1

struct node
{
        struct  node *lchild;
        int info;
        struct  node *rchild;
        int balance;
};

void inorder(struct node *ptr);
struct node *RotateLeft(struct node *pptr);
struct node *RotateRight(struct node *pptr);
struct node *insert(struct node *pptr,int ikey);
struct node *insert_left_check(struct node *pptr,int *ptaller);
struct node *insert_right_check(struct node *pptr,int *ptaller);
struct node *insert_LeftBalance(struct node *pptr);
struct node *insert_RightBalance(struct node *pptr);
```

```
struct node *del(struct node *pptr, int dkey);
struct node *del_left_check(struct node *pptr,int *pshorter);
struct node *del_right_check(struct node *pptr,int *pshorter);
struct node *del_LeftBalance(struct node *pptr,int *pshorter);
struct node *del_RightBalance(struct node *pptr,int *pshorter);

main()
{
        int choice,key;
        struct node *root = NULL;

        while(1)
        {
                printf("\n");
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Inorder Traversal\n");
                printf("4.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        printf("Enter the key to be inserted : ");
                        scanf("%d",&key);
                        root = insert(root,key);
                        break;
                 case 2:
                        printf("Enter the key to be deleted : ");
                        scanf("%d",&key);
                        root = del(root,key);
                        break;
                 case 3:
                        inorder(root);
                        break;
                 case 4:
                        exit(1);
                 default:
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
```

6.14.3 Insertion in an AVL tree

The basic insertion algorithm is similar to that of binary search tree. We will search for the position where the new node is to be inserted and then insert the node at its proper place. After insertion, the balance factors of some nodes might change, so we have to record these changes. We know that the permissible balance factors in an AVL tree are 1, 0, -1, so if the balance factor of any node becomes 2 or -2, then we have to do additional work to restore the property of AVL tree. Let us take an AVL tree (T) and see the effect of inserting some nodes in it.

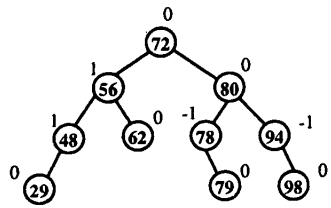


Figure 6.60 AVL tree (T)

We will insert the nodes 77, 58, 35, 99 in the tree T one by one and observe the change in balance factors.

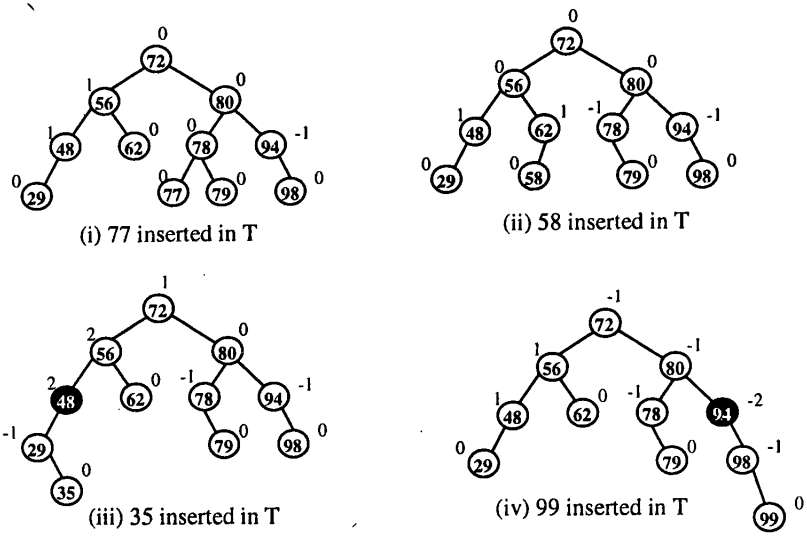


Figure 6.61

The nodes are inserted in appropriate place following the same procedure as in binary search tree. In all the four cases of figure 6.61, we can see that the balance factors of only ancestor nodes are affected i.e. only those nodes are affected which are in the path from inserted node to the root node. If the value of balance factor of each node on this path is -1, 0, or 1 after insertion, then the tree is balanced and has not lost its AVL tree property after the insertion of new node. Therefore, there is no need to do any extra work. If the balance factor of any node in this path becomes 2 or -2 then the tree becomes unbalanced. The first node on this path that becomes unbalanced(balance factor 2 or -2) is marked as the pivot node. So a pivot node is the nearest unbalanced ancestor of the inserted node.

In the first two cases(insertion of 77 and 58) the tree remains AVL after the insertion of new node, while in the last two cases(insertion of 35 and 99) the balance factors of some nodes become 2 or -2, thus violating the AVL property. In third case 48 is marked as the pivot node since it is the nearest unbalanced ancestor of the inserted node, similarly in fourth case 94 is marked as the pivot node.

After the insertion of new node in the tree, we examine the nodes in the path starting from newly inserted node to the root. For each node in this path, the following three cases can occur-

(A) Before insertion the node was balanced(0) and now it becomes left heavy(1) or right heavy(-1). The height of subtree rooted at this node changes and hence the balance factor of the parent of this node will change so we need to check the balance factor of the parent node. So in this case, we update the balance factor of the node and then check the balance factor of the next node in the path.

(B) Before insertion the node was left heavy(1) or right heavy(-1) and the insertion is done in the shorter subtree, so the node becomes balanced(0). We will update the balance factor of the node. The height of subtree rooted at this node does not change and hence the balance factors of the ancestors of this node will remain unchanged. Therefore, we can stop the procedure of checking balance factors i.e. there is no need to go upto the root and examine balance factors of all the nodes.

(C) Before insertion the node was left heavy(1) or right heavy(-1), and the insertion is done in the heavy subtree, so the node becomes unbalanced(2 or -2). Since the balance factor of 2 or -2 is not permitted in an AVL tree, we will not directly update the balance factor unlike the other two cases. This node is marked as the

pivot node and balancing is required. The balancing is done by performing rotations at this pivot node and updating the balance factors accordingly. The balancing is done in such a way that the height of the subtree rooted at pivot node is same before and after insertion. Therefore, the balance factors of ancestors will not change and we can stop the procedure of checking balance factors.

Now let us apply these cases in the four insertions that we have done-

(i) Insertion of 77 in tree T – Nodes to be checked are 78, 80 and 72. Firstly, node 78 is examined, before insertion it was right heavy and now it has become balanced, this is Case B, we'll change the balance factor to 0 and stop checking of balance factors. Balance factors of all the ancestors of this node will remain same as they were before insertion. This is because the height of the subtree rooted at 78 did not change after insertion.

(ii) Insertion of 58 in tree T – Nodes to be checked are 62, 56 and 72. Firstly, node 62 is examined, before insertion it was balanced and now it has become left heavy, this is Case A, so we'll change the balance factor to 1 and now check node 56. Node 56 was left heavy before insertion and now it became balanced(Case B), so we'll change the balance factor of node 56 to 0 and now we can stop procedure of checking balance factors.

(iii) Insertion of 35 in tree T – Nodes to be checked are 29, 48, 56 and 72. Firstly node 29 is examined, before insertion it was balanced and now it has become right heavy, this is Case A, so we'll change the balance factor to -1 and check the next node which is 48. This node was left heavy before insertion and insertion is done in its left subtree so now it has become unbalanced(pivot node), this is case C so now balancing will have to be done and after balancing we can stop checking of balance factors.

(iv) Insertion of 99 in T – Nodes to be checked are 98, 94, 80 and 72. Firstly node 98 is examined, before insertion it was balanced and now it has become right heavy, this is Case A, so we'll change the balance factor to -1 and check next node which is 94. This node was right heavy before insertion, and insertion is done in its right subtree so now it has become unbalanced(pivot node), this is case C so now balancing will have to be done and after balancing we can stop checking of balance factors.

This was an outline of the insertion process, now we will look how we can write the function for insertion. In binary search tree, we had written both recursive and non-recursive functions for insertion. Here we will write the recursive version because we can easily check the balance factors of ancestors in the unwinding phase. We will take recursive insert function of BST as the base and make some additions in it so that the tree remains balanced after insertion. Here is the function for insertion in an AVL tree-

```
struct node *insert(struct node *pptr,int ikey)
{
        static int taller;

        if(pptr==NULL) /*Base case*/
        {
                pptr = (struct node *) malloc(sizeof(struct node));
                pptr->info = ikey;
                pptr->lchild = NULL;
                pptr->rchild = NULL;
                pptr->balance = 0;
                taller = TRUE;
        }
        else if(ikey < pptr->info)      /*Insertion in left subtree*/
        {
                pptr->lchild = insert(pptr->lchild,ikey);
                if(taller==TRUE)
                        pptr = insert_left_check(pptr,&taller);
        }
        else if(ikey > pptr->info)      /*Insertion in right subtree*/
        {
                pptr->rchild = insert(pptr->rchild,ikey);
```

```
            if(taller==TRUE)
                pptr = insert_right_check(pptr,&taller);
        }
        else  /*Base Case*/
        {
            printf("Duplicate key\n");
            taller = FALSE;
        }

        return pptr;
}/*End of insert()*/
```

There are two recursive calls in this function. After the recursion stops, we will pass through each node in the path starting from inserted node to the root node i.e. in the unwinding phase we will pass through each ancestor of the inserted node.

The functions `insert_left_check()` and `insert_right_check()` are written after the recursive calls so they will be called in the unwinding phase. The function `insert_left_check()` will be called to check and update the balance factors when insertion in done in left subtree of the current node. If insertion is done in right subtree of the current node then the function `insert_right_check()` will be called for this purpose. These functions are not always called; they are called only when the flag `taller` is TRUE. Now let us see what this `taller` is and why are these functions called conditionally.

In the three cases that we have studied earlier, we know that we can stop checking if case B or case C occurs at any ancestor node. So `taller` is a flag which is initially set to true when the node is inserted and when any of the last two cases(B or C) are encountered we will make it false(inside `insert_left_check()` or `insert_right_check()`). The process of checking balance factors stops when `taller` becomes false.

Now we will try to understand the working of this function with the help of an example. Let us take the example of insertion of node 58 in tree T (figure 6.61(ii)). The recursive calls are shown in the figure 6.62.

There are 4 invocations of `insert()` and the recursion stops when `insert()` is called with `pptr` as NULL. In this case, a new node is allocated and its address is assigned to `pptr`. All the fields of this node are set to proper values. The value of flag `taller` is set to true. The value of `pptr` is returned and the unwinding phase begins.

The control returns to the previous invocation(3rd call) of `insert()` and since `taller` is true, the function `insert_left_check()` is called(insertion in left subtree of 62). Inside this function, the appropriate action will be taken according to the case that occurs. We have seen earlier that case A occurs at this point so inside `insert_left_check()` the balance factor of node 62 will be reset. We know that when case A occurs we do not stop the procedure of checking balance factors so the value of flag `taller` will remain true.

Now control returns to the previous invocation(2nd call) of `insert()` and since the value of `taller` is true, the function `insert_right_check()` is called(insertion in right subtree of 56). This time case B occurs so inside `insert_right_check()`, we'll reset the balance factor of node 56 and now we have to stop the process of checking balance factors. For this we will make the value of `taller` false inside the function `insert_right_check()`. This will ensure that the functions `insert_left_check()` and `insert_right_check()` will not be called in the remaining invocations of `insert()`.

Now control returns to the first invocation of `insert()` and since now the value of `taller` is false, the function `insert_left_check()` will not be called. After this the control returns to `main()`.

If the key to be inserted is already present, then it is not inserted in the tree and so there is no need of checking the balance factors in the unwinding phase. Hence, the value of `taller` is made false in the case of duplicate key.

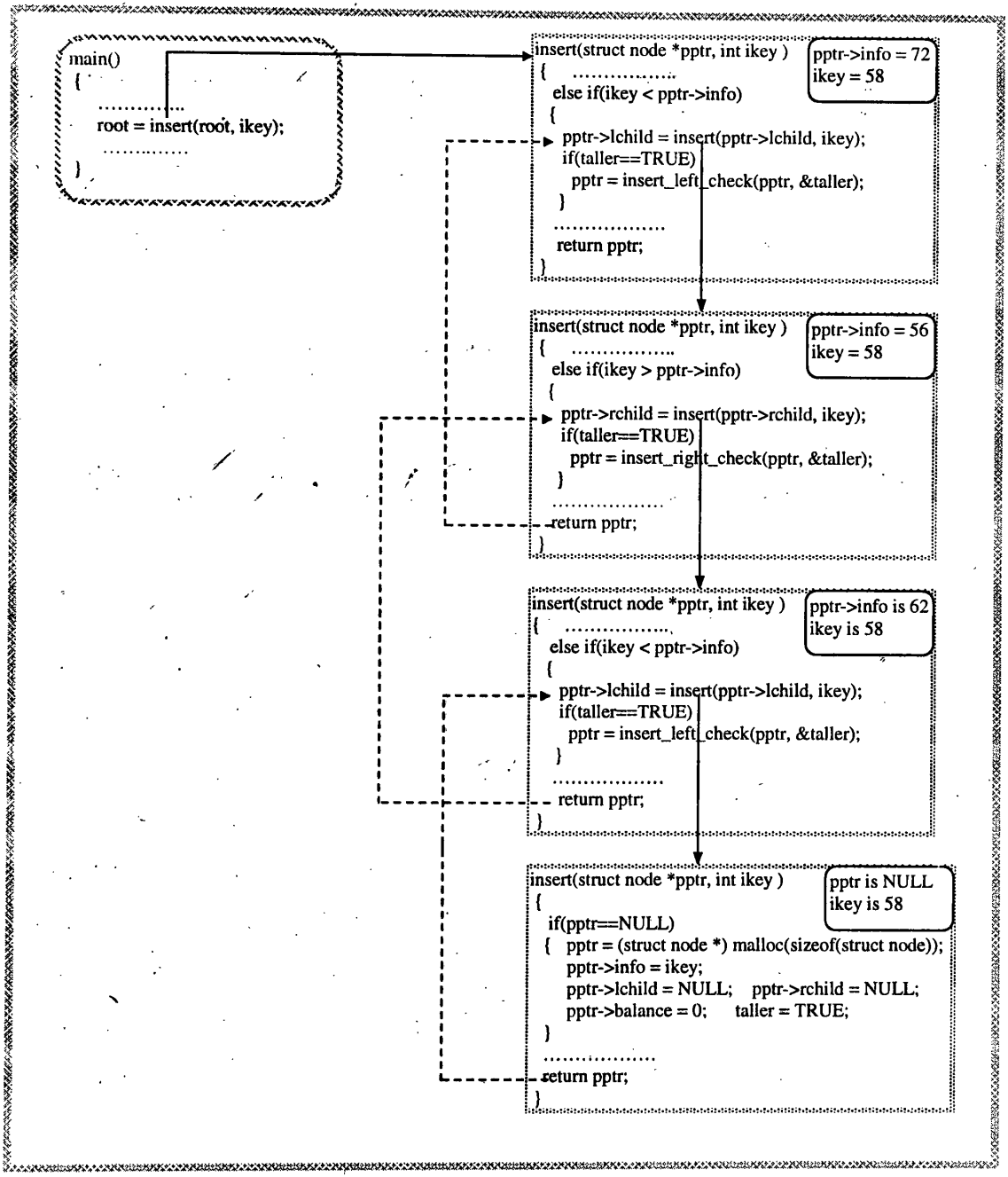


Figure 6.62

Now let us study in detail the different cases to be handled inside the functions `insert_left_check()` and `insert_right_check()`.

6.14.3.1 Insertion in left Subtree

When insertion is done in the left subtree of the current node, we use `insert_left_check()` to check and update the balance factors in unwinding phase. So before writing this function let us study the various cases involved in it. As stated earlier, initially the value of flag `taller` will be true.

P is the current node whose balance factor is being checked. PL and PR denote the left and right subtrees of P.. Insertion is done in left subtree of P. We will consider a subtree rooted at the node P.

Case L_A :

Before insertion : bf(P) = 0

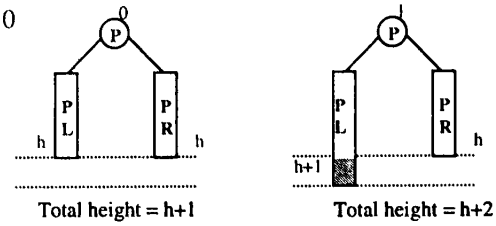


Figure 6.63

After insertion : bf(P) = 1

Height of the subtree rooted at node P has increased, this means that the balance factor of parent of node P will definitely change, so we need to continue the process of checking the balance factors. Hence, the value of `taller` remains TRUE.

Case L_B :

Before insertion : bf(P) = -1

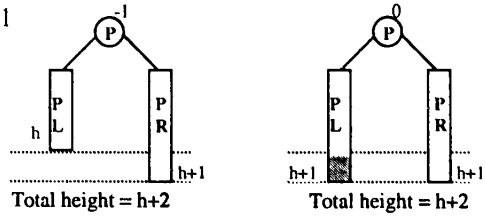


Figure 6.64

After insertion bf(P) = 0

Height of the subtree rooted at node P has not increased, this means that the balance factors of parent and ancestors of node P will not change, so we can stop the process of checking balance factors. Therefore, the value of `taller` is made FALSE.

Case L_C :

Before insertion : bf(P) = 1

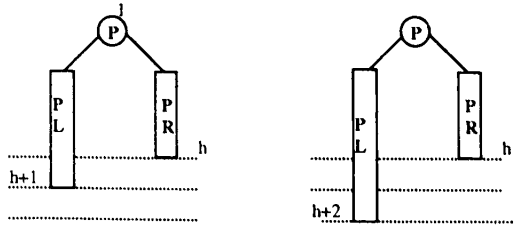


Figure 6.65

After insertion, the balance factor of P will become 2 and this is not a permissible value for balance factor in an AVL tree so we will not update it. The node P becomes unbalanced and it is the first node to become unbalanced so it is the pivot node. Now left balancing is required to restore the AVL property of the tree.

Now let us see how we can balance the tree in this case by performing rotations. We will further explore the left subtree of pivot node. Let A be the root of subtree PL, and AL, AR be left and right subtrees of A. Therefore, the subtree rooted at pivot node before insertion is-

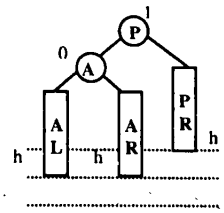


Figure 6.66

The balance factor of A before insertion will definitely be zero(1 or -1 is not possible), we'll discuss the reason for it afterwards. Now we can have two cases depending on whether the insertion is done in AL or AR.

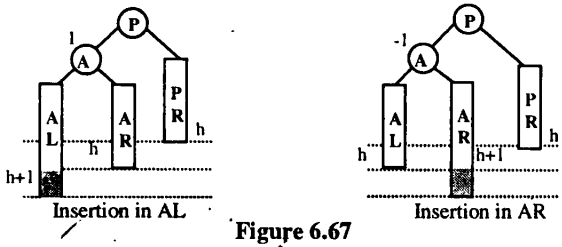


Figure 6.67

Currently we are checking the balance factor of the node P. It means that we have travelled till node A and updated all balance factors. So at this point, the balance factor of A will be 1 if insertion is done in AL or -1 if insertion is done in AR. The balance factor of pivot node is the same as was before insertion(it is still 1), we have not updated it to 2. Now we have two subcases depending on the balance factor of node A.

Case L_C1 :

Insertion done in left subtree of left child of node P(in AL)
Before insertion : bf(P) = 1, bf(A) = 0
After insertion, updated balance factor of A = 1
A single right rotation about P is performed

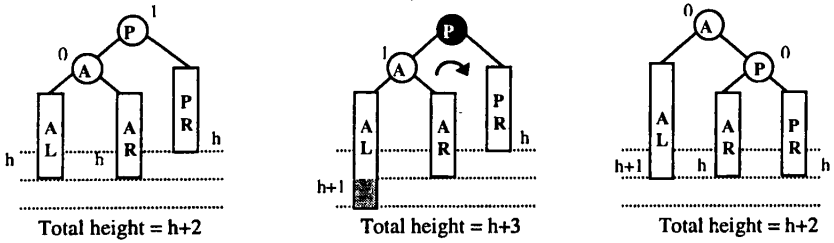


Figure 6.68

After balancing : bf(P) = 0, bf(A) = 0

The first figure shows the initial tree before insertion, second figure shows the tree after insertion at the point when we have updated balance factors till A and found that P has become unbalanced. The third figure shows the tree after rotation. After rotating, the balance factors of both pivot node P and node A become zero, and now node A is the new root of this subtree.

Here are two examples of case L_C1(right rotation).

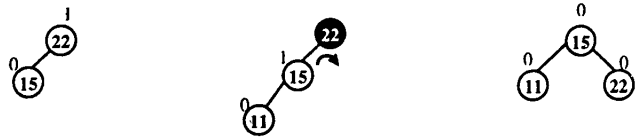


Figure 6.69 Insertion of 11

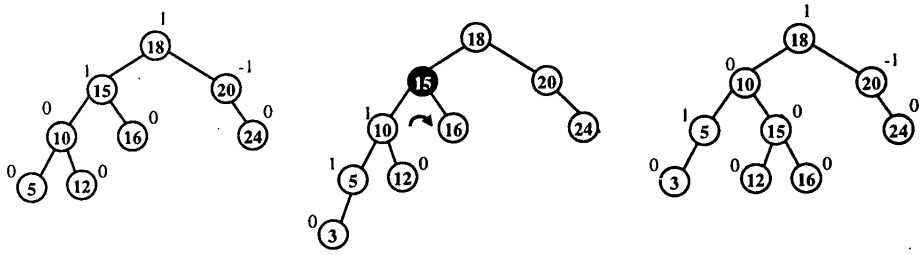


Figure 6.70 Insertion of 3

Case L_C2 :
Insertion done in right subtree of left child of node P (in AR)
Before insertion : bf(P) = 1, bf(A) = 0
After insertion updated balance factor of A = -1
LeftRight rotation performed

In this case, we need to explore the right subtree of A. Let B be the root of subtree AR, and let BL, BR be left and right subtrees of B. Before insertion, the balance factor of B will definitely be zero.

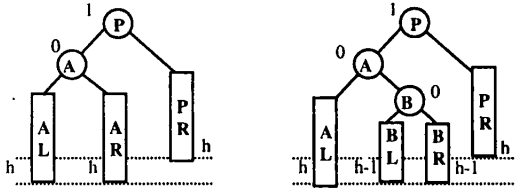


Figure 6.71

Here we can have two cases depending on whether insertion is done in BL or BR. It is possible that the subtree AR is empty before insertion, so in this case the newly inserted node is none other than B and this is our third case. So we have 3 subcases-

L_C2a : New node is inserted in BR (bf(B) = -1)
L_C2b : New node in inserted in BL (bf(B) = 1)
L_C2c : B is the newly inserted node (bf(B) = 0)

These 3 subcases are considered just to calculate the resulting balance factor of the nodes; otherwise the rotation performed is similar in all three cases.

A single rotation about the pivot node will not balance the tree so we have to perform double rotation here. First, we will perform a left rotation about node A, and then we will perform a right rotation about the pivot node P.

Case L_C2a :
New node is inserted in BR,
After insertion, updated balance factor of B = -1

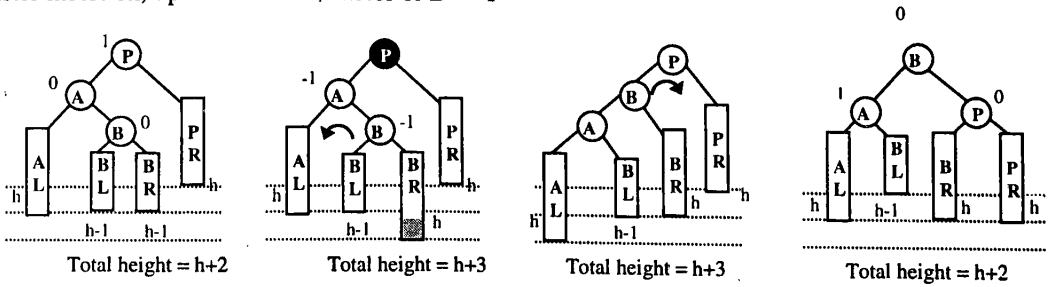


Figure 6.72

After balancing : bf(P) = 0, bf(A) = 1, bf(B) = 0
The first figure shows the initial tree before insertion, with AR further explored. The second figure shows the tree after insertion at the point when we have updated balance factors till A and found that P has become unbalanced. The third figure shows the tree after left rotation about A and the last figure shows the tree after right rotation about P.

Case L_C2b :
New node is inserted in BL
After insertion updated balance factor of B = 1

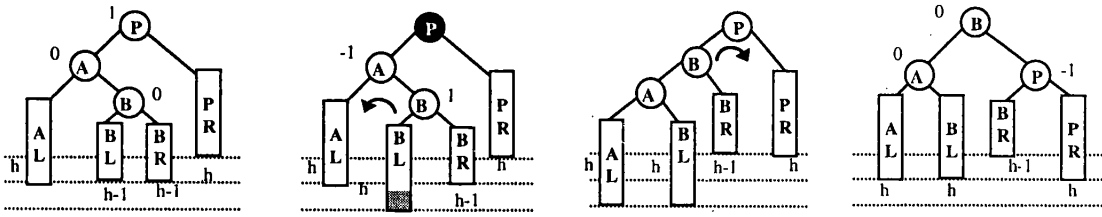


Figure 6.73

After balancing : bf(P) = -1, bf(A) = 0, bf(B) = 0

Case L_C2c :
B is the newly inserted node, so balance factor of B = 0
In the figure 6.71, if the newly inserted node is B, it means that AR was empty before insertion or we can say that value of h is zero. Since AL and PR also have height h, they will also be empty before insertion.

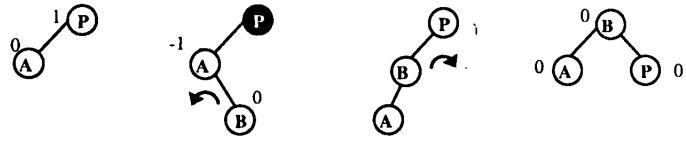


Figure 6.74

After balancing : bf(P) = 0, bf(A) = 0, bf(B) = 0
Here are two examples of case L_C2(LeftRight rotation).

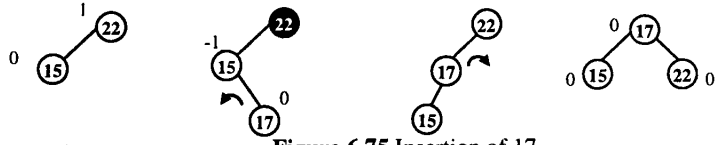


Figure 6.75 Insertion of 17

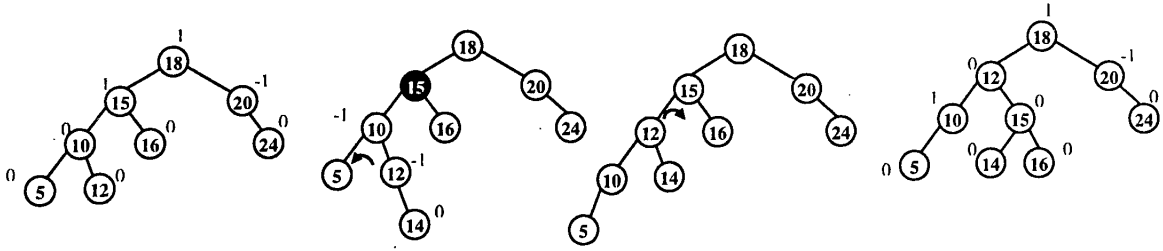


Figure 6.76 Insertion of 14

In case L_A, the value of `taller` remained true because the height of subtree rooted at P had increased and in case L_B the value of `taller` is made false because the height of subtree rooted at P did not change. In case L_C , the subtree rooted at P gets a new root but the height of this subtree remains same(in all subcases of L_C.

height before insertion and after balancing is h+2) and so taller is made false. The functions insert_left_check() is-

```
struct node *insert_left_check(struct node *pptr,int *ptaller)
{
        switch(pptr->balance)
        {
         case 0: /*Case L_A : was balanced*/
                pptr->balance = 1;      /*now left heavy*/
                break;
         case -1: /*Case L_B: was right heavy*/
                pptr->balance = 0;      /*now balanced*/
                *ptaller = FALSE;
                break;
         case 1: /*Case L_C: was left heavy*/
                pptr = insert_LeftBalance(pptr);       /*Left Balancing*/
                *ptaller = FALSE;
        }
        return pptr;
}/*End of insert_left_check()*/
```

The function insert_LeftBalance() is-

```
struct node *insert_LeftBalance(struct node *pptr)
{
        struct node *aptr,*bptr;
        aptr = pptr->lchild;
        if(aptr->balance==1)  /*Case L_C1 : Insertion in AL*/
        {
                pptr->balance = 0;
                aptr->balance = 0;
                pptr = RotateRight(pptr);
        }
        else            /*Case L_C2 : Insertion in AR*/
        {
                bptr = aptr->rchild;
                switch(bptr->balance)
                {
                case -1:                        /*Case L_C2a : Insertion in BR*/
                        pptr->balance = 0;
                        aptr->balance = 1;
                        break;
                case 1:                 /*Case L_C2b : Insertion in BL*/
                        pptr->balance = -1;
                        aptr->balance = 0;
                        break;
                case 0: /*Case L_C2c : B is the newly inserted node*/
                        pptr->balance = 0;
                        aptr->balance = 0;
                }
                bptr->balance = 0;
              - pptr->lchild = RotateLeft(aptr);
                pptr = RotateRight(pptr);
        }
        return pptr;
}/*End of insert_LeftBalance()*/
```

In the function insert_LeftBal() we have updated the balance factors before performing rotations. This is because rotations are done through functions RotateRight() and RotateLeft() and after calling these functions, pptr no longer points to node P but it points to the new root of the subtree.

The following information in the box summarizes all of the cases explained, and it includes the situation when the insertion is done in right subtree.

```
A new node is inserted and taller = TRUE
P is the node whose balance factor is being checked.

If Insertion in left subtree of the node P (Insertion in PL)
Case L_A :  If node P was balanced before insertion
                Node P becomes left heavy
Case L_B :  If node P was right heavy before insertion
                Node P becomes balanced, taller = FALSE
Case L_C : If node P was left heavy before insertion
                Node P becomes unbalanced, Left Balancing required, taller = FALSE
                P is the pivot node and its left child is A
                Case L_C1 : If insertion in left subtree of A ( in AL )
                                Right Rotation (right about P)
                Case L_C2 : If insertion in right subtree of A ( in AR )
                                LeftRight Rotation(left about A then right about P)
                                B is the right child of A
                                Case L_C2a : If insertion in right subtree of B ( in BR )
                                                bf(P) = 0, bf(A) = 1
                                Case L_C2b : If insertion in left subtree of B( in BL )
                                                bf(P) = -1, bf(A) = 0
                                Case L_C2c : If B is the newly inserted node.
                                                bf(P) = 0, bf(A) = 0

If Insertion in right subtree of the node P (Insertion in PR )
Case R_A : If node P was balanced before insertion
                Node P becomes right heavy
Case R_B : If node P was left heavy before insertion
                Node P becomes balanced, taller = FALSE
Case R_C : If node P was right heavy before insertion
                Node P becomes unbalanced, Right Balancing required, taller = FALSE
                P is the pivot node and its right child is A
                Case R_C1 : If Insertion in right subtree of A ( in AR )
                                Left Rotation (left about P)
                Case R_C2 : Insertion in left subtree of A ( in AL )
                                RightLeft Rotation (right about A then left about P)
                                B is the left child of A
                                Case R_C2a : If insertion in right subtree of B ( in BR )
                                                bf(P) = 1, bf(A) = 0
                                Case R_C2b : If insertion in left subtree of B ( in BL )
                                                bf(P) = 0, bf(A) = -1
                                Case R_C2c : If B is the newly inserted node.
                                                bf(P) = 0, bf(A) = 0
```

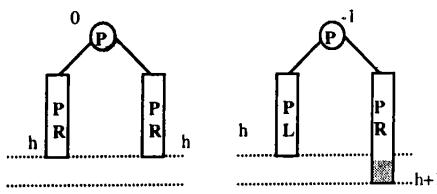
Figure 6.77 Insertion in an AVL Tree

6.14.3.2 Insertion in Right Subtree

In this case, insertion is done in the right subtree of P.

Case R_A :

Before insertion : bf(P) = 0



Total height = h+1 Total height = h+2

Figure 6.78

After insertion : bf(P) = -1
Height of the subtree rooted at pivot node changes, so `taller` remains TRUE.

Case R_B :
Before insertion : bf(P) = 1

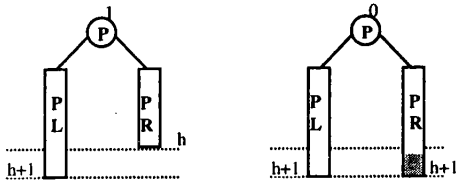


Figure 6.79

After insertion : bf(P) = 0
Height of the subtree rooted at pivot node does not change, so `taller` is made FALSE.

Case R_C :
Before insertion : bf(P) = -1

After insertion the node P will become unbalanced(-2) so P becomes the pivot node and balancing is required. We will explore the subtree PR and let A be the root and AL, AR the left and right subtrees of A. Like in case L_C, here also we have two cases depending on whether the insertion is done in AL or AR.

Case R_C1 :
Insertion done in right subtree of right child of node P (in AR).

After insertion updated balance factor of A = -1
Left rotation about node P is performed.

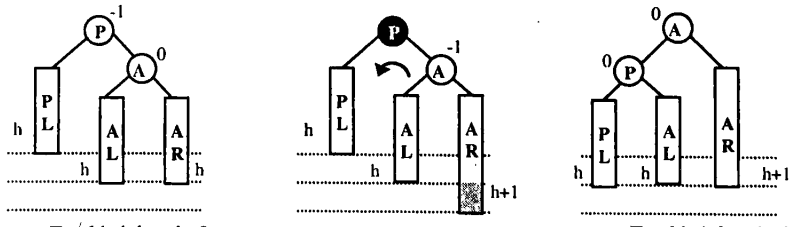


Figure 6.80

After balancing ; bf(P) = 0, bf(A) = 0

Here are two examples of case R_C1(left rotation).

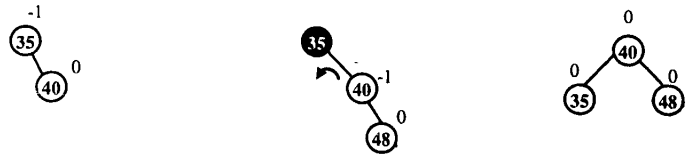


Figure 6.81 Insertion of 48

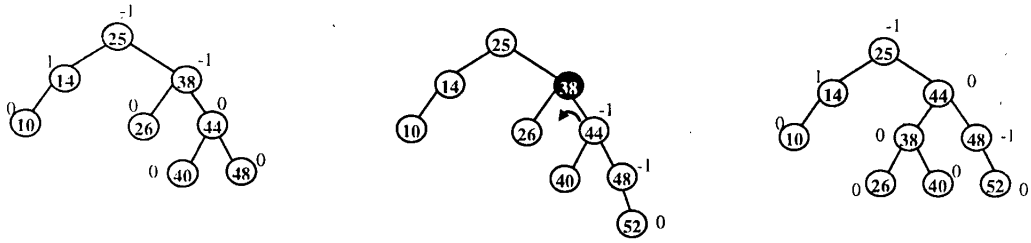


Figure 6.82 Insertion of 52

Case R_C2 :
Insertion done in left subtree of right child of node P (in AL).
After insertion updated balance factor of A = 1
RightLeft rotation performed.

We will further explore the left subtree of A. Let B be the root of subtree AL and let BL, BR be left and right subtrees of B. Before insertion, the balance factor of B will definitely be zero.

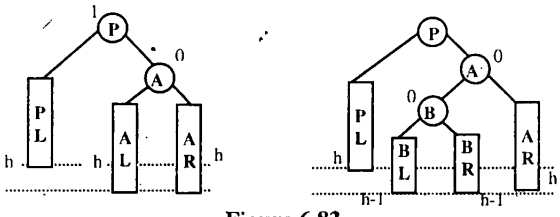


Figure 6.83

There can be 3 subcases-

R_C2a : New node is inserted in BR (bf(B) = -1)
R_C2b : New node in inserted in BL (bf(B) = 1)
R_C2c : B is the newly inserted node (bf(B) = 0)

Like case L_C2, here also double rotation is performed. First we will perform a right rotation about node A, and then we will perform a left rotation about the pivot node P. We can see that this is the mirror image of case of left to right rotation.

Case R_C2a :
New node is inserted in BR.
After insertion updated balance factor of B = -1

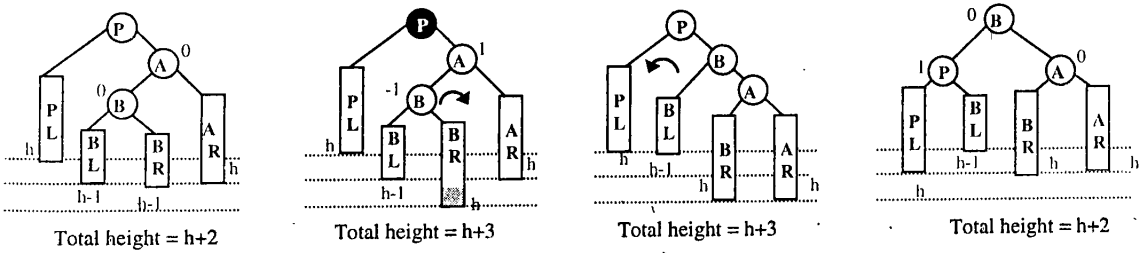


Figure 6.84

After balancing : bf(P) = 1, bf(A) = 0, bf(B) = 0

Case R_C2b :
New node is inserted in BL
After insertion updated balance factor of B = 1

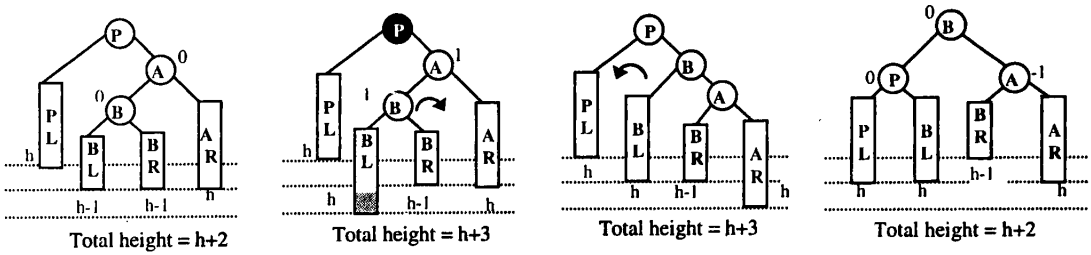


Figure 6.85

After balancing : bf(P) = 0, bf(A) = -1, bf(B) = 0

Case R_C2c :

B is the newly inserted node, so balance factor of B = 0

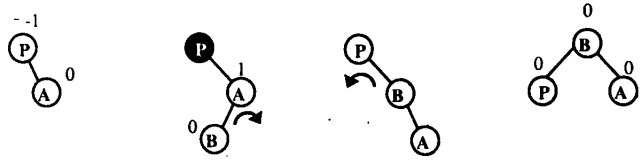


Figure 6.86

After balancing : bf(P) = 0, bf(A) = 0, bf(B) = 0

Here are two examples of case R_C2 (RightLeft rotation).

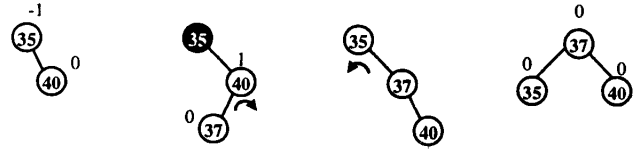


Figure 6.87 Insertion of 37

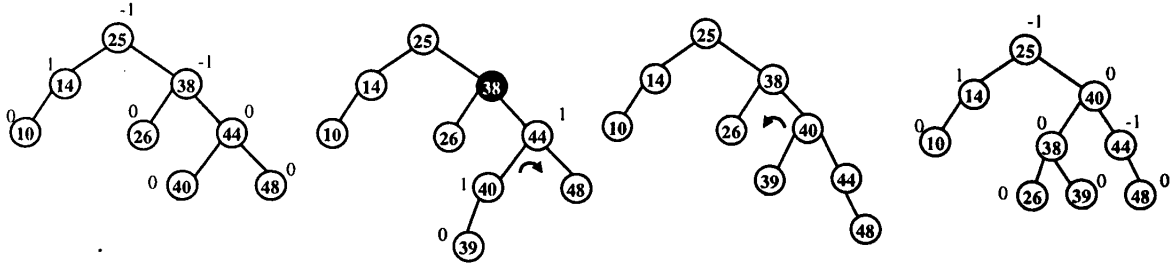


Figure 6.88 Insertion of 39

The function `insert_right_check()` is-

```
struct node *insert_right_check(struct node *pptr, int *ptaller)
{
        switch(pptr->balance)
        {
         case 0: /*Case R_A : was balanced*/
                 pptr->balance = -1;    /*now right heavy*/
                 break;
         case 1: /*Case R_B : was left heavy*/
                 pptr->balance = 0;     /*now balanced*/
```

```
                *ptaller = FALSE;
                break;
        case -1: /*Case R_C: Right heavy*/
                pptr = insert_RightBalance(pptr);     /*Right Balancing*/
                *ptaller = FALSE;
        }
        return pptr;
}/*End of insert_right_check()*/
```

The function `insert_RightBalance()` is-

```
struct node *insert_RightBalance(struct node *pptr)
{
        struct node *aptr, *bptr;
        aptr = pptr->rchild;
        if(aptr->balance==-1) /*Case R_C1 : Insertion in AR*/
        {
                pptr->balance = 0;
                aptr->balance = 0;
                pptr = RotateLeft(pptr);
        }
        else        /*Case R_C2 : Insertion in AL*/
        {
                bptr = aptr->lchild;
                switch(bptr->balance)
                {
                case -1:        /*Case R_C2a : Insertion in BR*/
                        pptr->balance = 1;
                        aptr->balance = 0;
                        break;
                case 1:         /*Case R_C2b : Insertion in BL*/
                        pptr->balance = 0;
                        aptr->balance = -1;
                        break;
                case 0: /*Case R_C2c : B is the newly inserted node*/
                        pptr->balance = 0;
                        aptr->balance = 0;
                }
                bptr->balance = 0;
                pptr->rchild = RotateRight(aptr);
                pptr = RotateLeft(pptr);
        }
        return pptr;
}/*End of insert_RightBalance()*/
```

In all the 4 cases of rotation (Right, RightLeft, Left, LeftRight), the height of the subtree rooted at pivot remains the same as was before insertion so the balance factors of ancestors of pivot remain unchanged, hence there is no need of checking balance factors of the ancestors. Only one single or double rotation is sufficient to balance the whole tree.

In the above discussion, we have assumed that the balance factors of A and B before insertion will definitely be zero. Let us see why this is so.

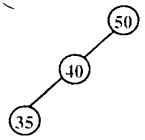
Whenever we find an unbalanced node(pivot node), it is definite that all the nodes in the path from newly inserted node to this pivot node had a balance factor of 0 prior to insertion. This is because if any node in this path had a balance factor of 1 or -1 before insertion, then after insertion either this node will become balanced(bf = 0, case B) or it will become unbalanced(bf = 2 or -2, case C). We know that in both these cases we stop checking balance factors. This is the reason for assuming the balance factors of A and B to be zero before insertion.

Let us take some numbers and construct an AVL tree from them.
50, 40, 35, 58, 48, 42, 60, 30, 33, 25

Insert 40

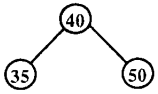
P : 50 Case L_A

Insert 35

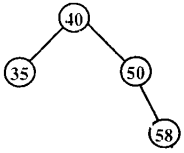


P : 40 Case L_A
P : 50 Case L_C1

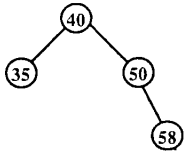
Rotate right about 50



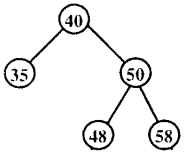
Insert 58



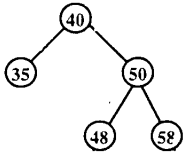
P : 50 Case R_A
P : 40 Case R_A



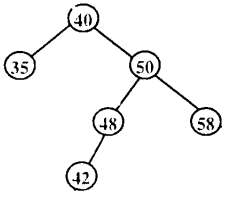
Insert 48



P : 50 Case L_B

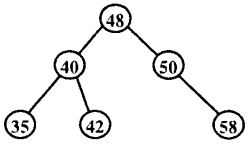


Insert 42

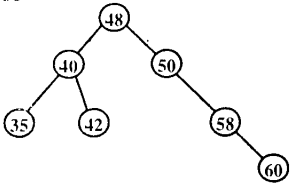


P : 48 Case L_A
P : 50 Case L_A
P : 40, Case R_C2b

Rotate right about 50
Rotate left about 40

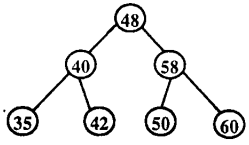


Insert 60

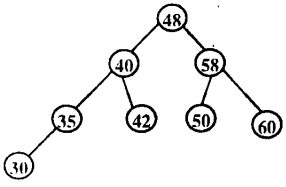


P : 58 Case R_A
P : 50, Case R_C1

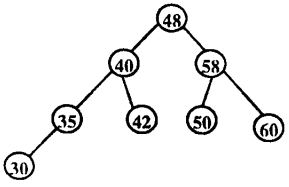
Rotate left about 50



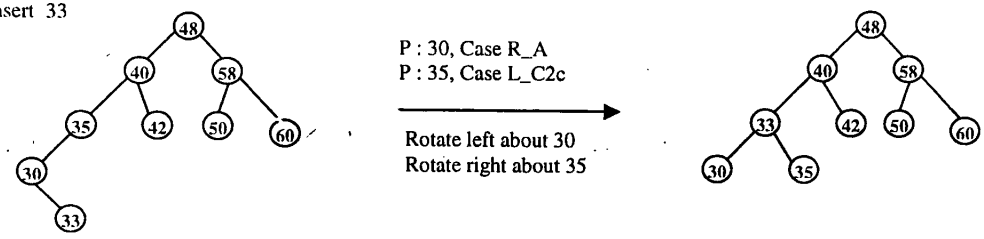
Insert 30



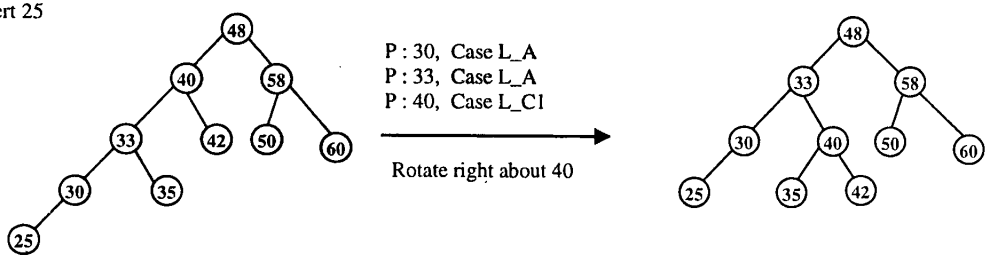
P : 35, Case L_A
P : 40, Case L_A
P : 48, Case L_A



Insert 33



Insert 25



There is no need to remember all the cases while inserting in an AVL tree. We just need to update the balance factors of the ancestor nodes and if any node becomes unbalanced we can perform the appropriate rotations.

6.14.4 Deletion in AVL tree

Deletion is performed using the same logic as in Binary search tree. After deletion, the tree may become unbalanced in some cases so we have to identify these cases and arrange for restoring the balance of the tree. We can take the deletion algorithm of BST as the base and modify it so that the AVL property is retained after deletion.

The node is deleted from the tree using recursive algorithm and in the unwinding phase we check the balance factors of the nodes in the path from deleted node to the root node. There can be three cases for each node in this path-

(A) Before deletion the node was balanced and after deletion it becomes left heavy or right heavy. We will update the balance factor of the node. In this case, the balance factors of the ancestors of this node will remain unchanged. Therefore, we can stop the procedure of checking balance factors.

(B) Before deletion, the node was left heavy or right heavy and the deletion is performed in the heavy subtree, so after deletion the node becomes balanced. We will update the balance factor of the node and then check the balance factor of the next node in the path.

(C) Before deletion, the node was left heavy or right heavy and the deletion is performed in the shorter subtree, so after deletion the node becomes unbalanced. In this case balancing is required which is performed using the same rotations that were done in insertion. There are 3 subcases in the case C. In one subcase, we will stop the procedure of checking balance factors while in the other two subcases we will continue our checking.

The function for deletion of a node from AVL tree is shown below-

```
struct node *del(struct node *pptr,int dkey)
{
        struct node *tmp,*succ;
        static int shorter;

        if(pptr==NULL) /*Base Case*/
        {
                printf("Key not present \n");
                shorter = FALSE;
```

```
            return(pptr);
        }
        if(dkey < pptr->info)
        {
            pptr->lchild = del(pptr->lchild,dkey);
            if(shorter == TRUE)
                pptr = del_left_check(pptr,&shorter);
        }
        else if(dkey > pptr->info)
        {
            pptr->rchild = del(pptr->rchild,dkey);
            if(shorter == TRUE)
                pptr = del_right_check(pptr,&shorter);
        }
        else /*dkey == pptr->info, Base Case*/
        {
            /*pptr has 2 children*/
            if(pptr->lchild!=NULL && pptr->rchild!=NULL)
            {
                succ = pptr->rchild;
                while(succ->lchild)
                    succ = succ->lchild;
                pptr->info = succ->info;
                pptr->rchild = del(pptr->rchild,succ->info);
                if(shorter == TRUE)
                    pptr = del_right_check(pptr,&shorter);
            }
            else
            {
                tmp = pptr;
                if(pptr->lchild != NULL) /*only left child*/
                    pptr = pptr->lchild;
                else if(pptr->rchild != NULL) /*only right child*/
                    pptr = pptr->rchild;
                else    /*no children*/
                    pptr = NULL;
                free(tmp);
                shorter = TRUE;
            }
        }
        return pptr;
}/*End of del()*/
```

This function is similar to the function for deletion in a BST, except for a few changes required to retain the balance of the tree after deletion. The flag `shorter` serves the same purpose as `taller` did in `insert()`. It is initially set to true when the node is deleted and it is made false inside `del_left_check()` or `del_right_check()`. The process of checking balance factors stops when `shorter` becomes false. When the item to be deleted is not found in the tree, then also `shorter` is made false because in this case we do not want any checking of balance factors in the unwinding phase.

6.14.4.1 Deletion from Left Subtree

P is the current node whose balance factor is being checked and PL, PR are its left and right subtrees. The deletion is done from the left subtree of P.

Case L_A :
Before deletion : bf(P) = 0

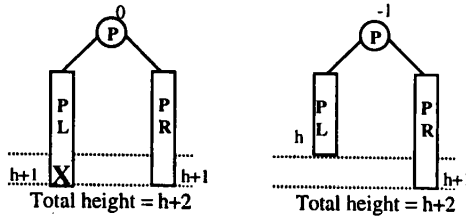


Figure 6.89

After deletion : bf(P) = -1
The height of the subtree rooted at P does not change after deletion so `shorter` is made FALSE.

Case L_B :
Before deletion : bf(P) = 1

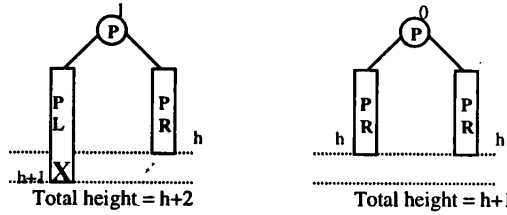


Figure 6.90

After deletion : bf(P) = 0
The height of the subtree rooted at P is decreased so `shorter` remains TRUE.

Case L_C :
Before deletion : bf(P) = -1

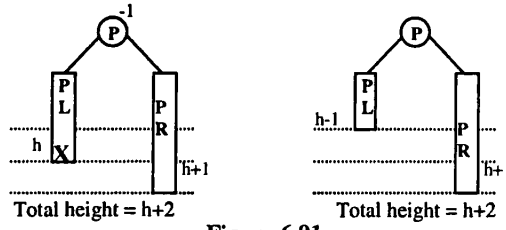


Figure 6.91

The node P will become unbalanced after deletion, so we will not update the balance factor of P directly. P becomes the pivot node and balancing is required. We will explore the right subtree of P. Let A be the root of PR, and let AL, AR be left and right subtrees of A.

Note that while insertion we had explored the subtree in which insertion was done, while here we are exploring the subtree other than the one from which node is deleted. So, if deletion is performed from left subtree of pivot node, then right balancing is required.

. The balance factors of only those nodes will be affected by deletion which lie in the path from deleted node to root node. Presently we are at the point when we have deleted the node and updated all the balance factors in PL and now we have found that P has become unbalanced. The balance factors of nodes in PR will remain unaffected after deletion since it is not in the path from deleted node to root. Node A is the root of PR so its balance factor will be same before and after deletion. Now we can have three cases depending on the three different values of balance factor of A.

Note that in insertion we had only two cases because there the balance factor of A before insertion was definitely 0. This was so because there A was in the path that was affected by insertion. In deletion, node A could have any of the three values(-1, 0, 1) before deletion since it is not in the path that is affected by deletion.

Case L_C1 :
Before deletion bf(P)= -1, bf(A) = 0
Left rotation about P is performed

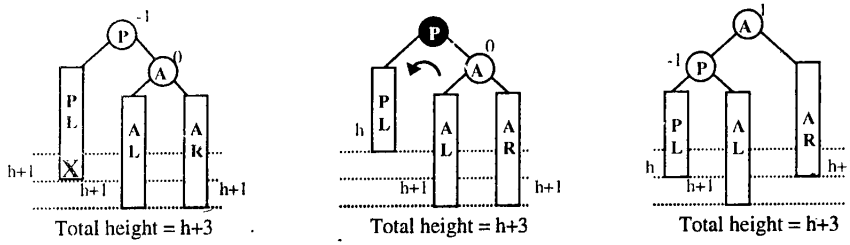


Figure 6.92

After balancing : bf(P) = -1, bf(A) = 1
The first figure shows the initial tree, second figure shows the tree after deletion at the point when we have updated all the balance factors in PL and found that P has become unbalanced. The third figure shows the tree after rotation. The height of the subtree rooted at P does not change, so `shorter` is made FALSE.

Case L_C2 :
Before deletion bf(P) = -1, bf(A) = -1
Left rotation about P performed

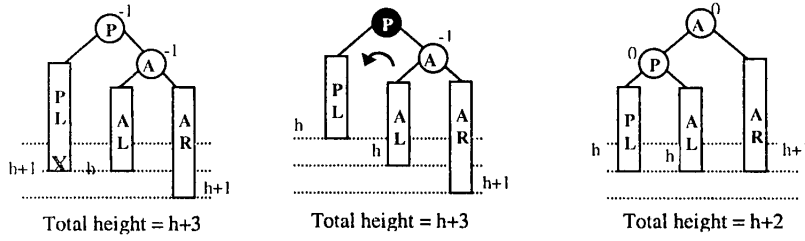


Figure 6.93

After balancing : bf(P) = 0, bf(A) = 0
In this case, height of the subtree has decreased so `shorter` remains TRUE.

Case L_C3 :
Before deletion : bf(P) = -1, bf(A) = 1
RightLeft rotation performed
In this case, a single rotation will not suffice so double rotation is performed. Now we will explore the left subtree of A and let B be the root node of AL.

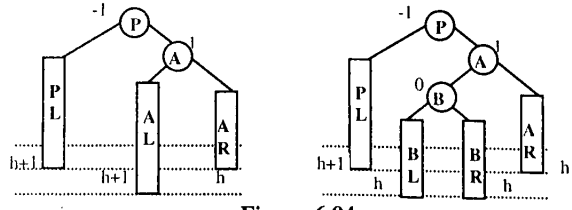


Figure 6.94

We can have three subcases depending on the three different balance factors of B. The rotation performed is same in all the three cases(RightLeft rotation) but the resulting balance factors of P and A are different.

Case L_C3a :
Before deletion : bf(P) = -1, bf(A) = 1, bf(B) = 0

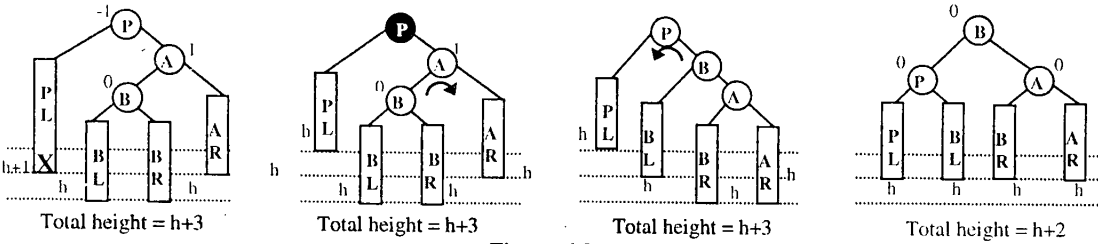


Figure 6.95

After balancing : bf(P) = 0, bf(A) = 0, bf(B) = 0

Case L_C3b :

Before deletion : bf(P) = -1, bf(A) = 1, bf(B) = 1

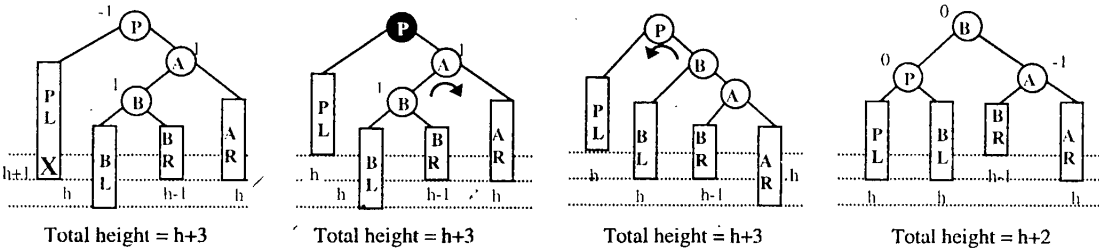


Figure 6.96

After balancing : bf(P) = 0, bf(A) = -1, bf(B) = 0

Case L_C3c :

Before deletion : bf(P) = -1, bf(A) = 1, bf(B) = -1

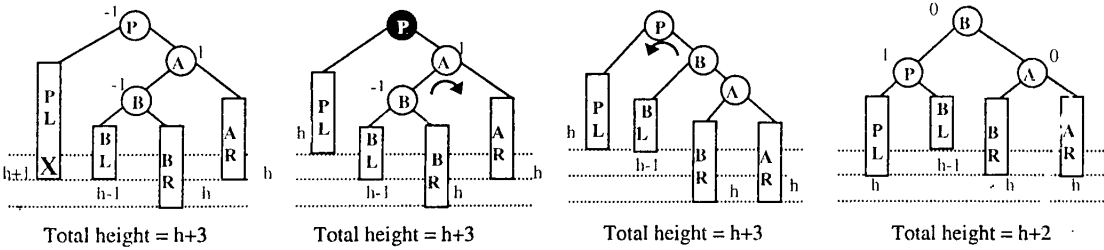


Figure 6.97

After balancing : bf(P) = 1, bf(A) = 0, bf(B) = 0

The height of the subtree decreases so the value of shorter remains TRUE in case L_C3.

The functions `del_left_check()` and `del_RightBalance()` are given below-

```
struct node *del_left_check(struct node *pptr,int *pshorter)
{
        switch(pptr->balance)
        {
                case 0: /*Case L_A : was balanced*/
                        pptr->balance = -1;     /*now right heavy*/
                        *pshorter = FALSE;
                        break;
                case 1: /*Case L_B : was left heavy */
                        pptr->balance = 0;      /*now balanced*/
                        break;
                case -1: /*Case L_C : was right heavy*/
                        /*Right Balancing*/
```

```
                        pptr = del_RightBalance(pptr, pshorter);
        }
        return pptr;
}/*End of del_left_check()*/
struct node *del_RightBalance(struct node *pptr,int *pshorter)
{
        struct node *aptr, *bptr;
        aptr = pptr->rchild;
        if(aptr->balance == 0) /*Case L_C1*/
        {
                pptr->balance = -1;
                aptr->balance = 1;
                *pshorter = FALSE;
                pptr = RotateLeft(pptr);
        }
        else if(aptr->balance == -1)  /*Case L_C2*/
        {
                pptr->balance = 0;
                aptr->balance = 0;
                pptr = RotateLeft(pptr);
        }
        else                                    /*Case L_C3*/
        {
                bptr = aptr->lchild;
                switch(bptr->balance)
                {
                        case 0:                 /*Case L_C3a*/
                                pptr->balance = 0;
                                aptr->balance = 0;
                                break;
                        case 1:                 /*Case L_C3b*/
                                pptr->balance = 0;
                                aptr->balance = -1;
                                break;
                        case -1:                        /*Case L_C3c*/
                                pptr->balance = 1;
                                aptr->balance = 0;
                }
                bptr->balance = 0;
                pptr->rchild = RotateRight(aptr);
                pptr = RotateLeft(pptr);
        }
        return pptr;
}/*End of del_RightBalance()*/
```

6.14.4.2 Deletion from Right Subtree

Case R_A :

Before deletion : bf(P) = 0

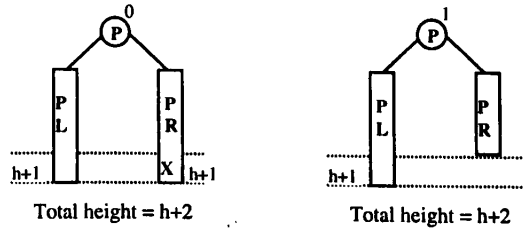


Figure 6.98

After deletion : bf(P) = 1
shorter = FALSE

Case R_B :
Before deletion : bf(P) = -1

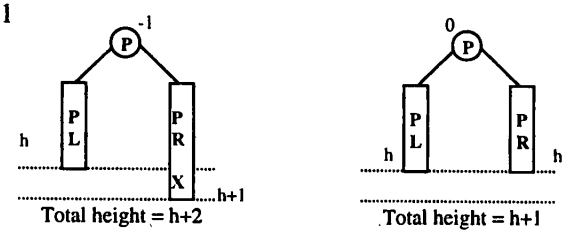


Figure 6.99

After deletion : bf(P) = 0
shorter remains TRUE

Case R_C :
Before deletion : bf(P) = 1
Node P becomes unbalanced, balancing required.

Case R_C1 :
Before deletion : bf(P) = 1, bf(A) = 0
Right rotation about P performed

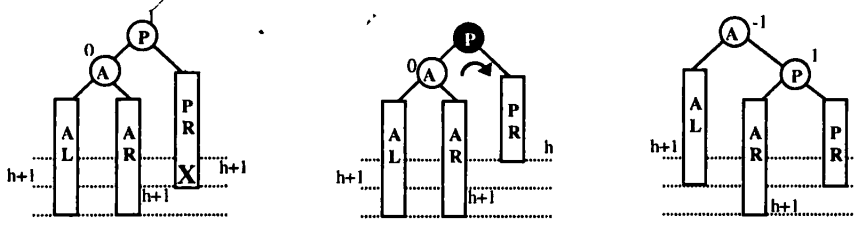


Figure 6.100

After balancing : bf(P) = 1, bf(A) = -1
shorter =FALSE

Case R_C2 :
Before deletion : bf(P) = 1, bf(A) = 1
Right rotation performed

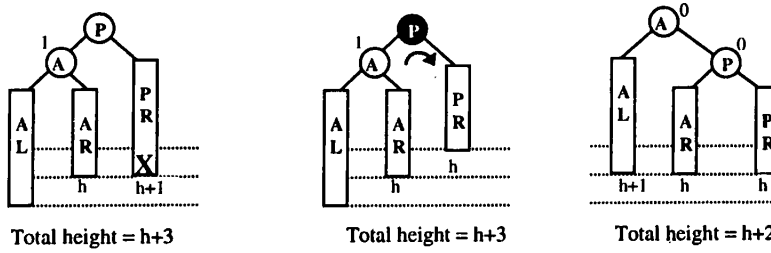


Figure 6.101

After balancing : bf(P) = 0, bf(A)=0
shorter remains TRUE.

Case R_C3 :
Before deletion : bf(P) = 1, bf(A) = -1
LeftRight rotation performed

Case R_C3a :
Before deletion : bf(P) = 1, bf(A) = -1, bf(B) = 0

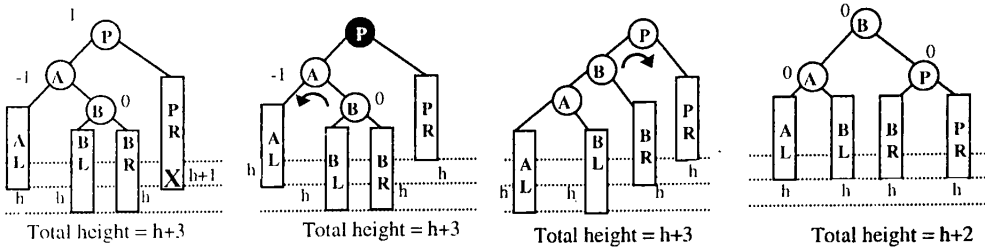


Figure 6.102

After balancing : bf(P) = 0, bf(A)=0, bf(B) = 0

Case R_C3b :
Before deletion : bf(P) = 1, bf(A) = -1, bf(B) = 1

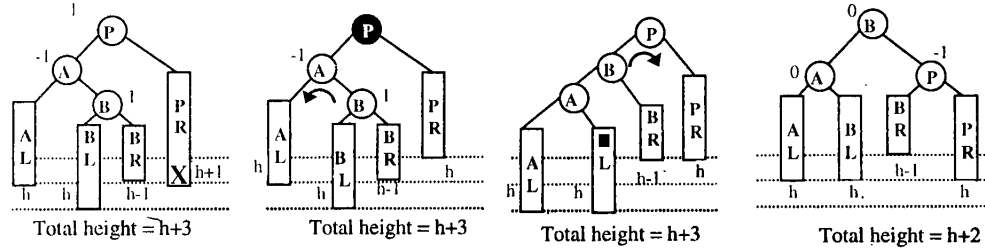


Figure 6.103

After balancing : bf(P) = -1, bf(A)=0, bf(B) = 0

Case R_C3c :
Before deletion : bf(P) = 1, bf(A) = -1, bf(B) = -1

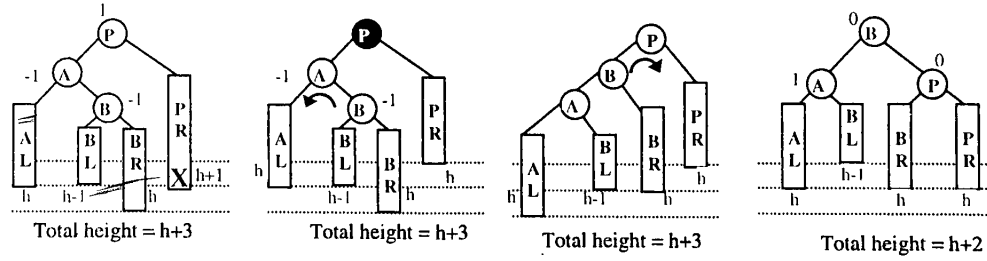


Figure 6.104

After balancing : bf(P) = 0, bf(A)=1, bf(B) = 0

The functions `del_right_check()` and `del_LeftBalance()` are-

```
struct node *del_right_check(struct node *pptr, int *pshorter)
{
        switch(pptr->balance)
        {
                case 0:         /*Case R_A : was balanced*/
                        pptr->balance = 1;      /*now left heavy*/
                        *pshorter = FALSE;
                        break;
                case -1: /*Case R_B : was right heavy*/
                        pptr->balance = 0;      /*now balanced*/
                        break;
```

```
                case 1: /*Case R_C : was left heavy*/
                        pptr = del_LeftBalance(pptr,pshorter );/*Left Balancing*/
        }
        return pptr;
}/*End of del_right_check()*/
struct node *del_LeftBalance(struct node *pptr,int *pshorter)
{
        struct node *aptr,*bptr;
        aptr = pptr->lchild;

        if(aptr->balance == 0)  /*Case R_C1*/
        {
                pptr->balance = 1;
                aptr->balance = -1;
                *pshorter = FALSE;
                pptr = RotateRight(pptr);
        }
        else if(aptr->balance == 1) /*Case R_C2*/
        {
                pptr->balance = 0;
                aptr->balance = 0;
                pptr = RotateRight(pptr);
        }
        else    /*Case R_C3*/
        {
                bptr = aptr->rchild;
                switch(bptr->balance)
                {
                        case 0:      /*Case R_C3a*/
                                pptr->balance = 0;
                                aptr->balance = 0;
                                break;
                        case 1:         /*Case R_C3b*/
                                pptr->balance = -1;
                                aptr->balance = 0;
                                break;
                        case -1:      /*Case R_C3c*/
                                pptr->balance = 0;
                                aptr->balance = 1;
                }
                bptr->balance = 0;
                pptr->lchild = RotateLeft(aptr);
                pptr = RotateRight(pptr);
        }
        return pptr;
}/*End of del_LeftBalance()*/
```

All the cases and subcases of deletion are shown in the figure 6.105.

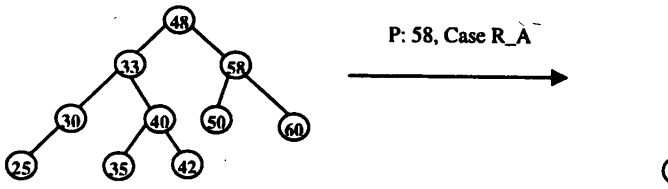
Note that in the case of insertion after making one rotation(single or double), the variable `taller` was made FALSE and there was no need to proceed further. From the discussion of deletion, we can see that `shorter` is not always made FALSE after rotations. For example, rotations are performed in cases L_C2, L_C3, R_C2, R_C3 but `shorter` is not made FALSE. This means that even after performing rotation we may have to proceed and check the balance factors of other nodes. So, in deletion one rotation may not suffice to balance the tree unlike the case of insertion. In the worst case, each node in the path from deleted node to root node may need balancing.

```
A new node is deleted and shortrer = TRUE
P is the node whose balance factor is being checked.

If deletion from left subtree of the node P (deletion from PL)

  Case L_A :  If node P was balanced before deletion
                    Node P becomes right heavy, shorter = FALSE
  Case L_B :  If node P was left heavy before deletion
                    Node P becomes balanced
  Case L_C : If node P was right heavy before deletion
                    Node P becomes unbalanced, Right Balancing required
                    A is right child of P
                    Case L_C1 : If bf(A) = 0
                                        Left Rotation, shorter = FALSE
                                        bf(P) = -1, bf(A) = 1
                    Case L_C2 : If bf(A) = -1
                                        Left Rotation
                                        bf(P) = 0, bf(A) = 0
                    Case L_C3 : If bf(A) = 1
                                        RightLeft Rotation (right about A, left about P)
                                        B is the left child of A
                                        BL and BR are left and right subtrees of B
                                        Case L_C3a : if bf(B) == 0
                                                            bf(P) = 0, bf(A) = 0
                                        Case L_C3b : If bf(B) == 1
                                                            bf(P) = 0, bf(A) = -1
                                        Case L_C3c : If bf(B) == -1
                                                            bf(P) = 1, bf(A) = 0

  If deletion from right subtree of the node P (deletion from PR)

  Case R_A : If node P was balanced before deletion
                    Node P becomes left heavy, shorter = FALSE
  Case R_B : If node P was right heavy before deletion
                    Node P becomes balanced
  Case R_C : If node P was left heavy before deletion
                    Node P becomes unbalanced, left Balancing required
                    A is left child of P
                    Case R_C1 : If bf(A) = 0
                                        Right Rotation, shorter = FALSE
                                        bf(P) = 1, bf(A) = -1
                    Case R_C2 : If bf(A) = 1
                                        Right Rotation
                                        bf(P) = 0, bf(A) = 0
                    Case R_C3 : If bf(A) = -1
                                        LeftRight Rotation ( left about A, right about P)
                                        B is the right child of A
                                        Case R_C3a : If bf(B) == 0
                                                            bf(P) = 0, bf(A) = 0
                                        Case R_C3b : If bf(B) == 1
                                                            bf(P) = -1, bf(A) = 0
                                        Case R_C3c : If bf(B) == -1
                                                            bf(P) = 0, bf(A) = 1
```

Figure 6.105

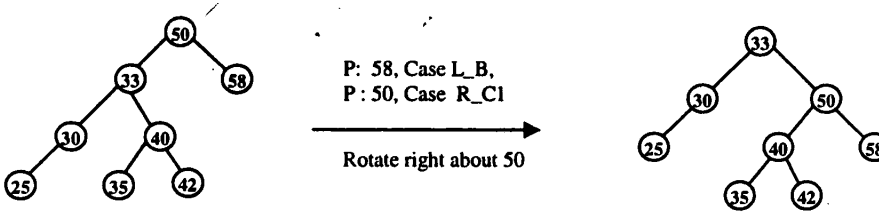
Now we will take an AVL tree and delete some nodes from it one by one-

Delete 60

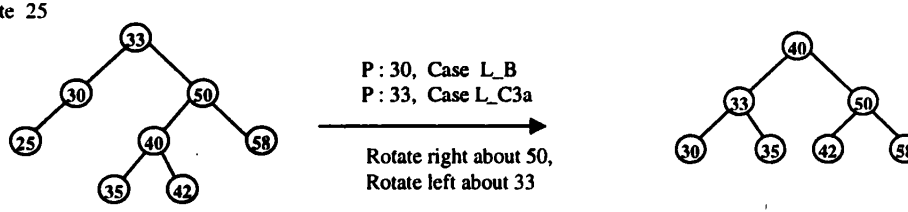


Delete 48

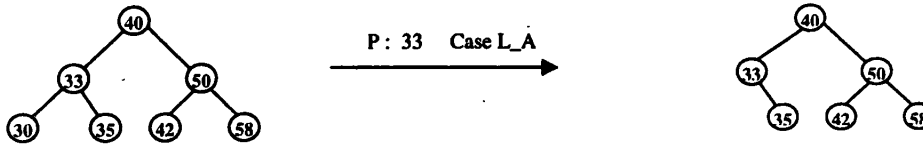
48 is not a leaf node, so its inorder successor 50 is copied at its place and the node 50 is deleted from the tree.



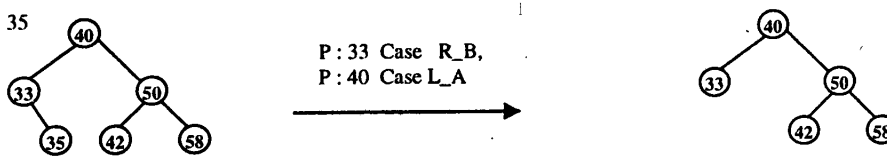
Delete 25

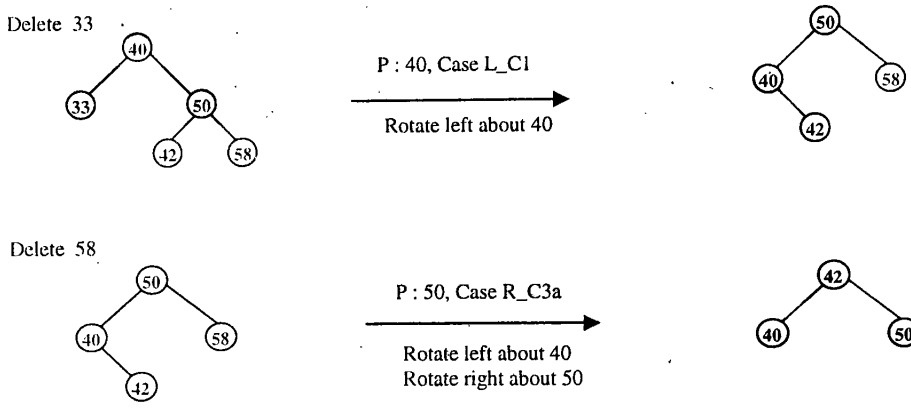


Delete 30



Delete 35





6.15 Red Black Trees

A red black tree is a balanced binary search tree in which every node is colored either red or black. Properties of this tree can be explained best in terms of an extended binary tree. Recall that an extended binary tree is a tree in which all NULL links are replaced by special nodes that are called external nodes. The properties of a red black tree are-

(P1) Root is always black.

(P2) All external nodes are black.

(P3) A red node cannot have red children; it can have only black children. There is no restriction on the color of the children of a black node, i.e. a black node can have red or black children.

(P4) For each node N, all paths from node N to external nodes contain the same number of black nodes. This number is called black height of that node.

Black height of a node N is defined as the number of black nodes from N to an external node(without counting node N). The black height of root node is called the black height of the tree. The black height of all external nodes is 0.

Following are two examples of red black trees; the black height of each node is shown outside the node.

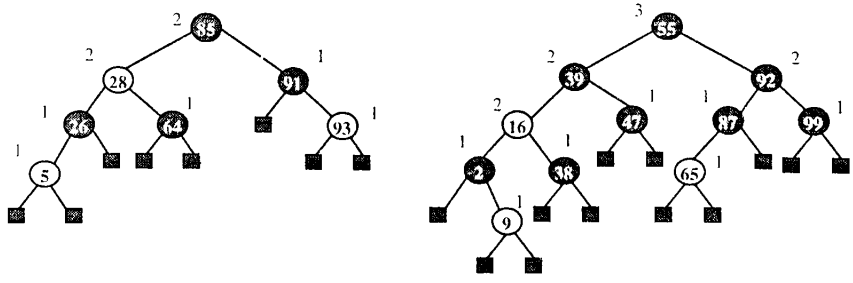


Figure 6.106

The root node is black in both trees so property P1 is satisfied for both of them. None of the red nodes has a red child so property P3 is also satisfied for them. The property P4 is also satisfied for both the trees. The black height of first tree is 2 and the black height of second tree is 3. Now let us see some trees that do not satisfy these properties.

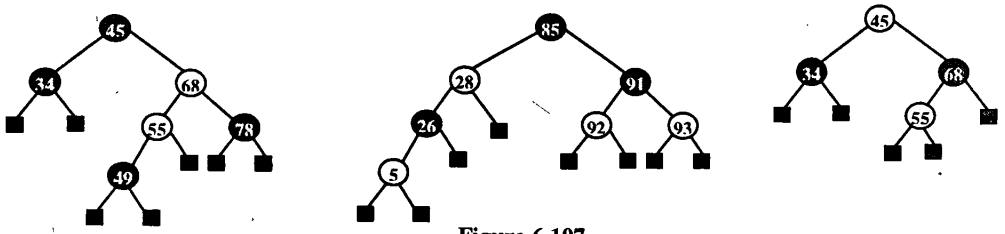


Figure 6.107

In the first tree, property P3 is violated since node 68 is red and it has a red child. We will refer to this problem as double red problem. In the second tree, property P4 is violated for nodes 28 and 85. We will refer to this problem as black height problem. In the third tree, property P1 is violated since root is red.

According to property P4, all paths from a node N to any external node have same number of black nodes. This is true for the root node as well i.e. any path from root node to any external node will have same number of black nodes. Now suppose the black height of root node is k.

If a path from root to external node has all the black nodes, then it will be the smallest possible path. If in a path the black and red nodes occur alternately, then it will be the longest possible path. Since black height of root node is k, each path will have k black nodes. Therefore, the total nodes in shortest possible path will be k and total nodes in longest possible path will be 2k. Thus we see that no path in a red black tree can be more than twice the another path of the tree. Hence a red black tree always remains balanced. The height of a red black tree with n internal nodes is at most $2\log_2(n+1)$.

In the structure of a node of red black tree, we will take an extra member color that can take two values, red or black. We will also need to maintain a parent pointer that will point to the parent of the node.

```
struct node
{
        enum {black,red} color;
        int info;
        struct node *lchild;
        struct node *rchild;
        struct node *parent;
};
```

Before proceeding further, let us define some terms that we will be using in the insertion and deletion operations.

Grandparent - Grandparent of a node is the parent of the parent node.

Sibling - Sibling of a node is the other child of parent node.

Uncle - Uncle is the sibling of parent node.

Nephews - Nephews of a node are the children of the sibling node.

Near Nephew - If node is left child, then left child of sibling node is near nephew and if node is right child, then right child of sibling node is near nephew.

Far Nephew - The nephew which is not the near nephew is the far nephew. In the figure 6.108, the nephew that appears closer to the node is the near nephew and the other one is far nephew.

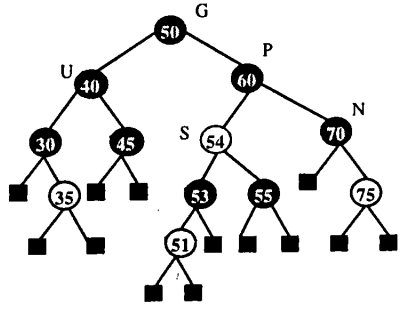


Figure 6.108

In the tree given in figure 6.108, we have marked the relatives of node 70. Its parent is node 60, grandparent is node 50, uncle is node 40, sibling is node 54, near nephew is node 55 and far nephew is node 53.

6.15.1 Searching

Since red black tree is a binary search tree, the procedure of searching is same as in a binary search tree. Searching operation does not modify the tree so there is need of any extra work.

6.15.2 Insertion

Initially the key is inserted in the tree following the same procedure as in a binary search tree. Each node in red black tree should have a color, so now we have to decide upon the color of this newly inserted node. If we color this new node black, then it is certain that property P4 will be violated i.e. we'll have black height problem(you can verify this yourself). If we color the new node red, then property P1 or P3 may be violated. The property P1 will be violated if this new node is root node, and property P3 will be violated(double red problem) if new node's parent is red Thus when we give red color to the new node, we will have some cases where no property will be violated after insertion while if we color it black it is guaranteed that property P4 will be violated. Therefore, the better option is to color the newly inserted node red.

We have decided to color the new node red, so we will never have violation of property P4. If the node is inserted in an empty tree, i.e. new node is the root node, then property P1 will be violated and to fix this problem, we will have to color the new node black. If new node's parent is black then no property will be violated. If new node's parent is red, then we will have a double red problem and now we will study this case in detail.

We have two situations, depending on whether the parent is left child or right child. In both situations, we have symmetric cases. The outline of all the subcases in this case is given below.

```
Parent is Red
        -Parent is left child,
                Case L_1 : Uncle is red
                                Recolor
                Case L_2 : Uncle is Black
                        Case L_2a : Node is right child
                                Rotate left about parent and transform to case L_2b
                        Case L_2b : Node is left child
                                Recolor and Rotate right about grandparent.
        -Parent is right Child
                Case R_1 : Uncle is red
                        Recolor
                Case R_2 : Uncle is black
                        Case R_2a : Node is left child
                                Rotate right about parent and transform to case R_2b
                        Case R_2b : Node is right child
                                Recolor and Rotate left about grandparent.
```

Now let us take all the cases one by one. In the figures, we have marked current node as N, its parent as P, its grandparent as G, and its uncle as U. Initially the current node is the newly inserted node, but we will see that in some cases, the double red problem is moved upwards and so the current node also moves up.

The left and right rotations performed are similar to those performed in AVL trees.

Case L_1 : Parent is left child and Uncle is red

In this case, we just recolor the parent and uncle to black and recolor grandparent to red. The following figures show this recoloring. The same recoloring is done if the node is right child.

Figure 6.109

Looking at the recolored figures it seems that now we do not have any double red problem, but remember that this is only a part of the tree and the node D might have a parent and grandparent. If the parent of node D is black, then we are done but if the parent of node D is red, then again we have a double red problem at node D. We have just moved the double red problem one level up.

Now the grandparent node is the node that needs to be checked for double red problem. So after recoloring, we make the grandparent as the new current node, we have marked it N in the recolored figure. Any of the 6 cases(L_1, L_2a, L_2b, R_1, R_2a, R_2b) may be applicable to this new current node.

It might happen that we repeatedly encounter this case(or case R_1) and we move the double red problem upwards by recoloring and ultimately we end up coloring the root red. In that case, we can just color the root black and our double red problem would be removed (insertion of 40 and insertion of 51 given in the examples of insertion).

Case L_2 : Uncle is black and rotation needs to be done.

Case L_2a : Parent is left child, Uncle is black and Node is right child

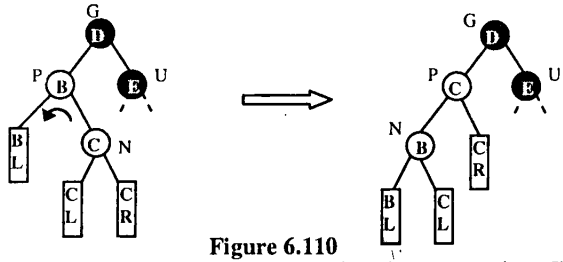


Figure 6.110

This case is transformed to case L_2b by performing a rotation. We perform a left rotation about the parent node B, and after rotation node C becomes the parent node, and node B becomes the current node. Now parent is left child, node is left child and uncle is black and this is case L_2b.

Case L_2b : Parent is left child, Uncle is black and Node is left child

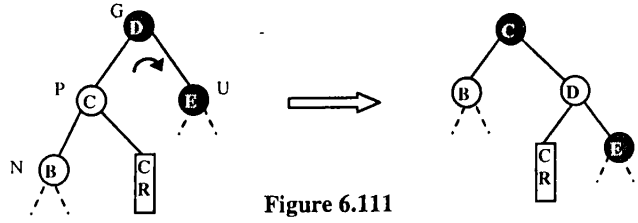


Figure 6.111

In this case parent is recolored black, grandparent is recolored red and a right rotation is performed about the grandparent node and this removes the double red problem.

The other three cases occur when parent is right child, they are symmetric to the previous three cases and only the figures of these cases are given.

Case R_1 : Parent is right child and uncle is red

Figure 6.112

Case R_2 : Uncle is black

Case R_2a : Parent is right child, uncle is black and node is left child

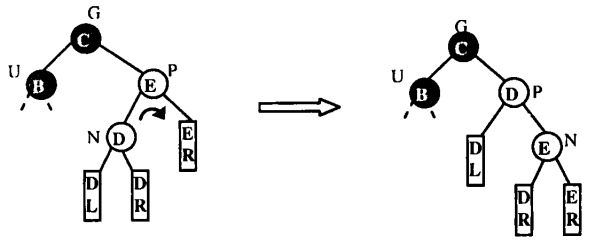


Figure 6.113

Case R_2b : Parent is right child, uncle is black and node is right child

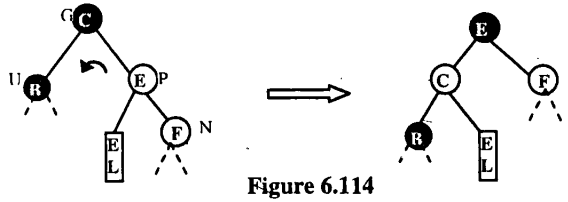
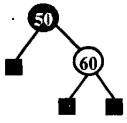


Figure 6.114

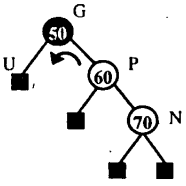
Now we will see examples of inserting some nodes in an initially empty red black tree. The explanation of each insertion is given at the end. While inserting, we will first check the color of the parent, if it is black there is nothing to be done. If parent is red, we have a double red problem and we will check the color of uncle. If uncle is red, we will recolor and move up, and if uncle is black, we will perform appropriate rotations and recoloring.

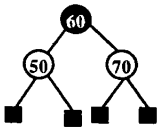
Insert 50

Insert 60

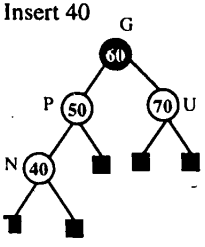


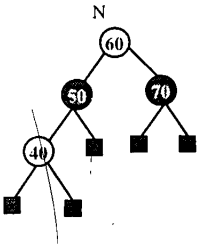
Insert 70

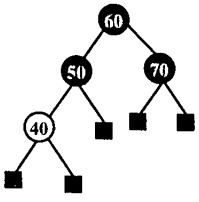




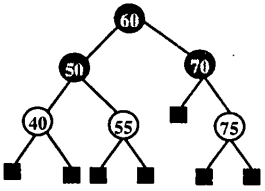
Insert 40



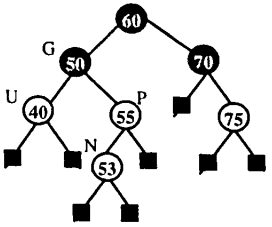


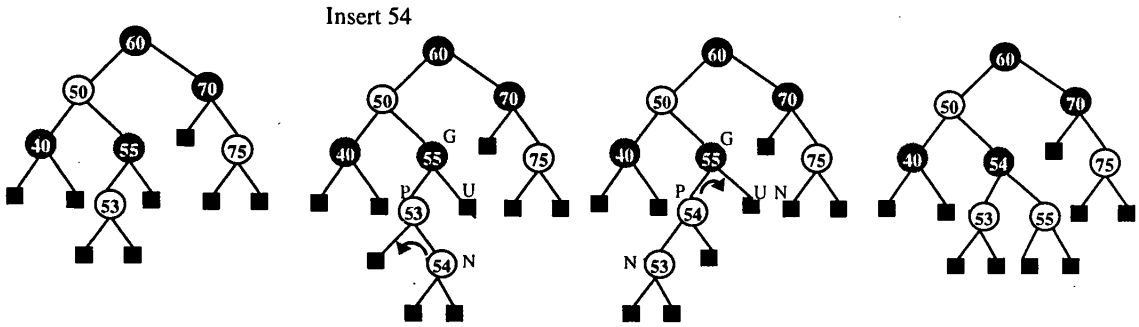


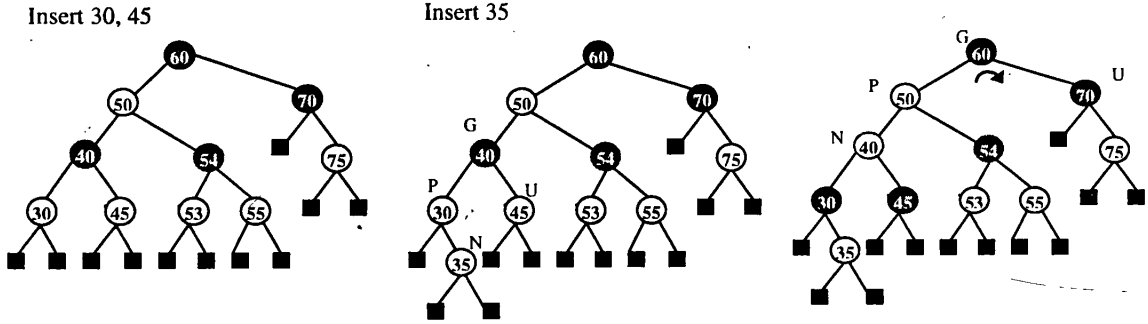
Insert 55, 75

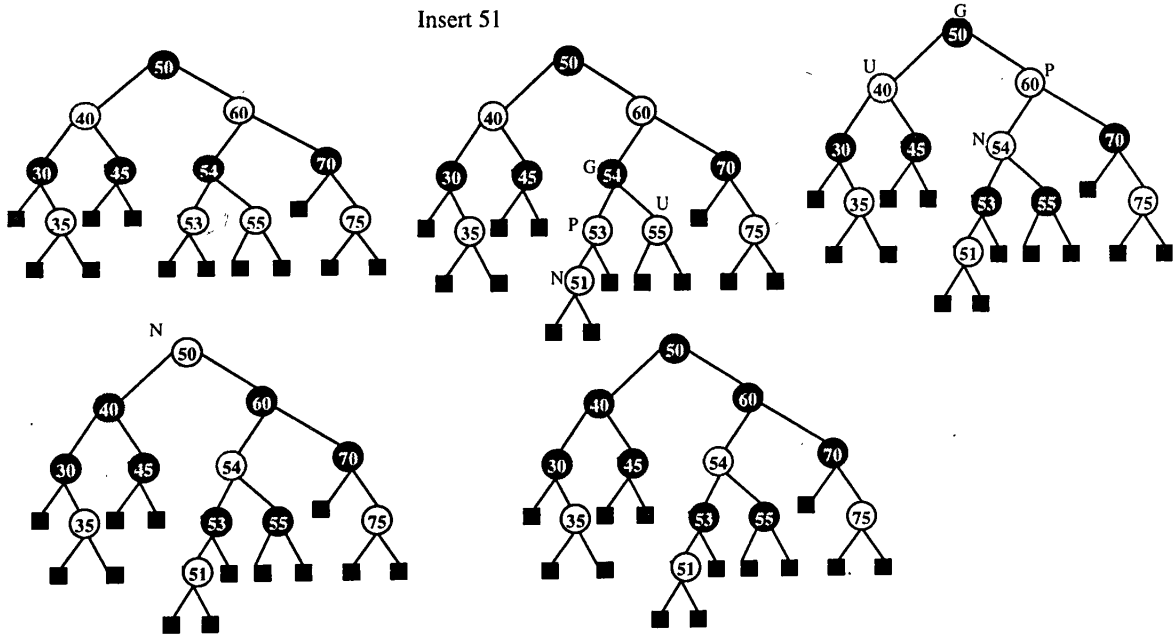


Insert 53









Insertion of 50 - Newly inserted node is the root node, so color it black.
Insertion of 60 - Parent is black so no violation of property.
Insertion of 70 - Parent is red, double red problem.
Uncle is black, parent is right child, node is right child(case R_2b).
Recolor and Rotate left about grandparent(50).
Insertion of 40 - Parent is red, double red problem. Uncle is red, recolor. Now node 60 is the current node.
We have colored the root red so color it black and we are done.

Insertion of 55 - Parent is black so no violation of property.
Insertion of 75 - Parent is black so no violation of property.
Insertion of 53 - Parent is red, double red problem. Uncle is red, recolor. Now node 50 is the current node.
Parent of 50 is black so we are done.
Insertion of 54 - Parent is red, double red problem.
Uncle is black, parent is left child, node is right child (case L_2a).
Rotate left about parent node and transform to case L_2b.
Recolor and rotate right about grandparent.
Insertion of 30 - Parent is black so no violation of property.
Insertion of 45 - Parent is black so no violation of property.
Insertion of 35 - Parent is red, double red problem. Uncle is red, recolor. Now node 40 is the current node.
Parent(50) is red, double red problem.
Uncle is black, parent is left child, node is left child(case L_2b).
Recolor and rotate right about grandparent.
Insertion of 51 - Parent is red, double red problem. Uncle is red, recolor. Now node 54 is the current node.
Parent is red, double red problem. Uncle is red, recolor. Now node 50 is the current node.
We have colored the root red so color it black and we are done.

6.15.3 Deletion

Initially the deletion of node is performed following the same procedure as in a binary search tree. We know that in a BST, if the node to be deleted has two children, then its information is replaced by the information of inorder successor and then the successor is deleted. The successor node will have either no child or only right child. Thus the case when node to be deleted has two children, is reduced to the case when node to be deleted has only one right child or no child.

Therefore, in our discussion, we will take the case of only those nodes that have only one right child or no child. This means that node to be deleted is always a parent of an external node i.e. node to be deleted has either one external node as a left child or two external nodes as children.

Suppose A is the node that has to be deleted, then we can think of six cases some of which are not possible.

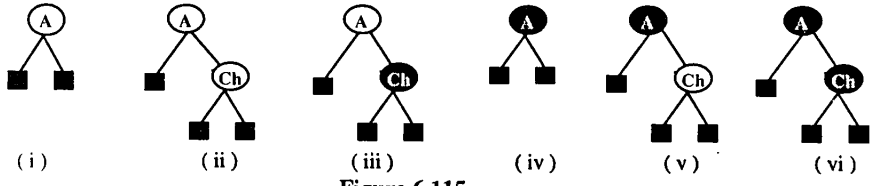


Figure 6.115

(i) A is red and has no children
No property violated and the tree remains red black after deleting node A.

(ii) A is red, and its only child is red
Impossible case, because it means that there was a double red problem in the tree.

(iii) A is red, and its only child is black
Impossible case, because this means node A was violating property P4.

(iv) A is black and has no children
In this case, after deletion the black height property would be violated and we need to restore this property. There are many subcases to be considered which we will study in detail.

(v) A is black, and its only child is red
In this case, the node A is deleted and its red child takes its place. This might create a double red problem if A had a red parent. The black height problem will definitely be introduced. Both these problems can be removed just by coloring this child black.

(vi) A is black, and its only child is black

Impossible case, because this means node A was violating property P4.

So, the three possible cases are (i), (iv) and (v). The actions to be taken in these three possible cases are-

- In Case (i), Node is red with no children
  Delete the node
- In Case (iv) Node is black with no children
  Delete the node and restore the black height property.
- In Case (v), Node is black and has a red child
  Delete the node and color the child black

Now we will study in detail the case (iv), when node to be deleted is a black node without any children. We will take N as the current node, i.e. it is the root of the subtree which is one black node short. Initially N will be the external node that replaced the node to be deleted and as we proceed we might move the current node upwards i.e. we might move the black height problem upwards. There can be different cases depending on the color of the sibling, nephews and parent.

Case 1 : N's sibling is Red

Case 2 : N's sibling is black, both nephews are black.
  Case 2a : Parent is Red
  Case 2b : Parent is Black

Case 3 : N's sibling is black, at least one nephew is red
  Case 3a : Far nephew is black, other nephew will be red.
  Case 3b : Far nephew is red, other nephew may be either red or black.

If node is left child, then we will name the cases as L_1, L_2a, L_2b, L_3a, L_3b and if node is right child then we'll have symmetric cases R_1, R_2a, R_2b, R_3a, R_3b.

Case L_1 : N's sibling is Red
In this case, parent and both nephews will definitely be black(by property P3).

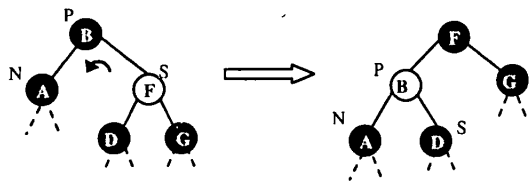


Figure 6.116

A left rotation is performed and the new sibling is D which is black, so this case is now converted to case 2 or case 3.

Case L_2 : N's sibling and both nephews are black

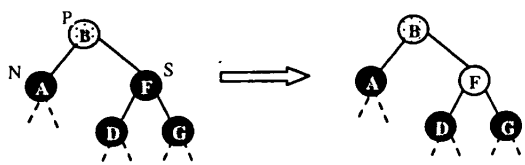


Figure 6.117

Node B is shaded which means that parent can be of either color. In this case, we color the sibling red. After that, we take different steps depending on the color of the parent.

Case L_2a : Parent is red
If parent is red, then after coloring the sibling red we have a double red problem. To remove this double red problem we color the parent black.

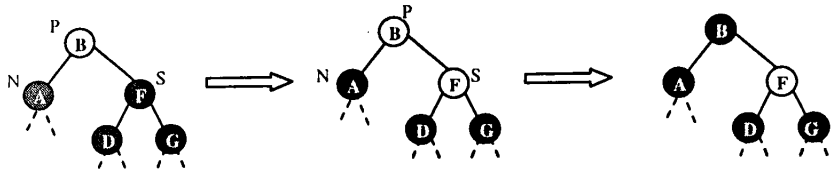


Figure 6.118

In the initial figure, subtree rooted at node A was one black node short, in the final figure we have introduced a black node in its path so now it not a black node short. Since we colored node F red, the black height of other subtree did not change.

Thus by removing the double red problem, we have removed the problem of shorter black height also and there is no need to proceed further. Note that when we enter case 2 from case 1 then we will enter case 2a only and not case 2b.

Case L_2b : Parent is black

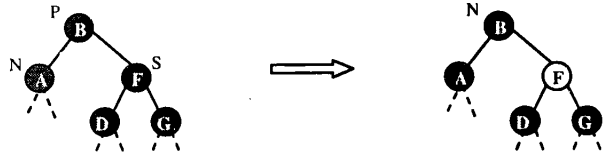


Figure 6.119

In this case the sibling is colored red and now the subtree rooted at A and subtree rooted at F both are one black node short. So we can say that the subtree rooted at B is one black node short. Now we make node B the current node and any of the cases may apply to this node. Thus, in this case we have moved the problem of shorter black height upwards.

Case L_3 : Sibling is black, at least one nephew is red

Case L_3a : Sibling is black, Far nephew is black, other nephew will be red.
This case is converted to case L_3b by performing a right rotation.

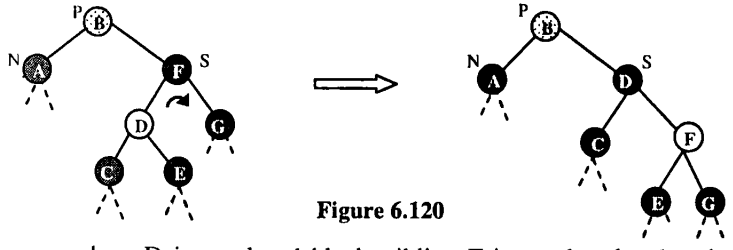


Figure 6.120

The near nephew D is recolored black, sibling F is recolored red and a right rotation is performed about sibling node. After rotation, node A's far nephew is node F which is red and so this case is converted to case L_3b.

Case L_3b : Sibling is black, Far nephew is red, other nephew may be either red or black.

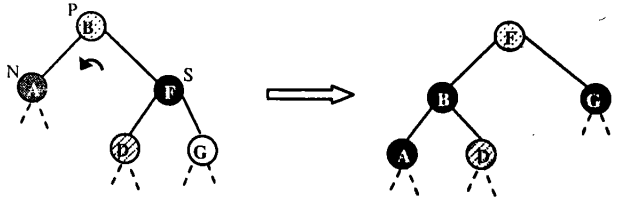


Figure 6.121

The parent node B and far nephew G are recolored black and node F is given the color of node B. Then a left rotation is performed about parent node and the black height problem is solved.

The figures of other symmetric cases are given next.

Case R_1 : N's sibling is Red

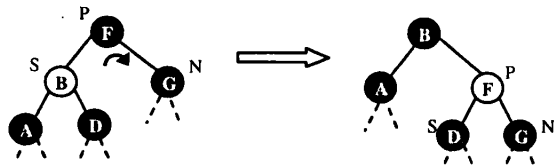


Figure 6.122

Case R_2a : N's sibling and both nephews are black, parent is red

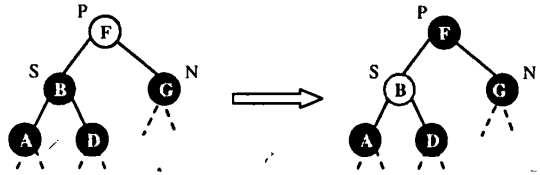


Figure 6.123

Case R_2b : N's sibling and both nephews are black, parent is black

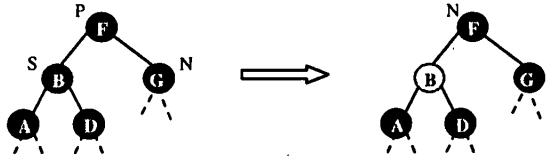


Figure 6.124

Case R_3a : Sibling is black, Far nephew is black, other nephew will be red.

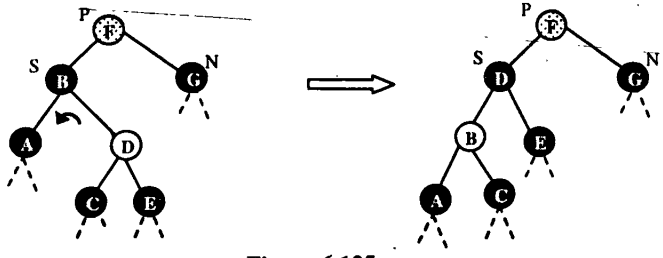


Figure 6.125

Case R_3b : Sibling is black, Far nephew is red, other nephew may be either red or black.

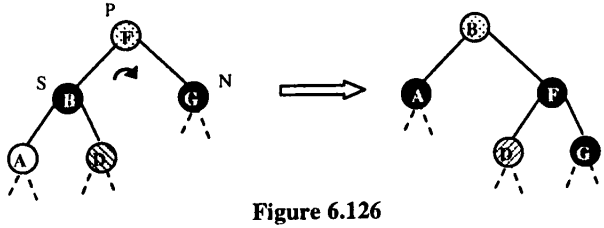
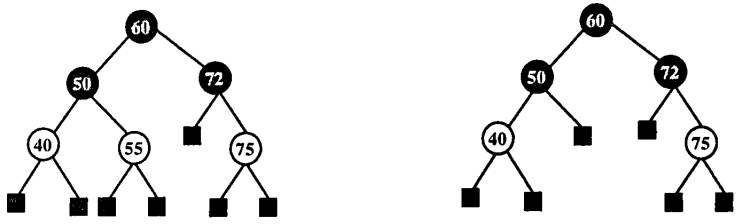


Figure 6.126

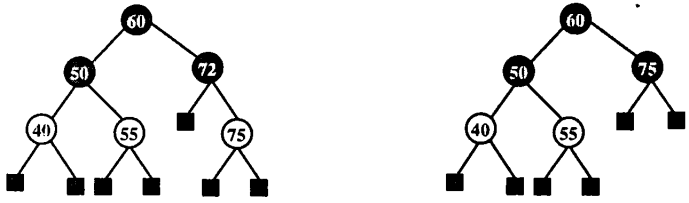
Now we will see some examples of deletion.

(i) Delete 55



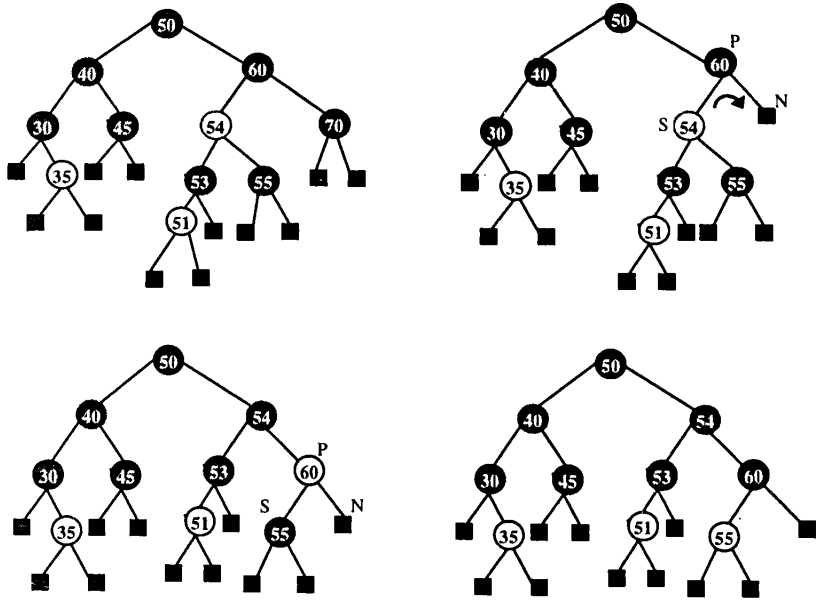
Here the node to be deleted is a red node so there is no violation of any property after deletion.

(ii) Delete 72



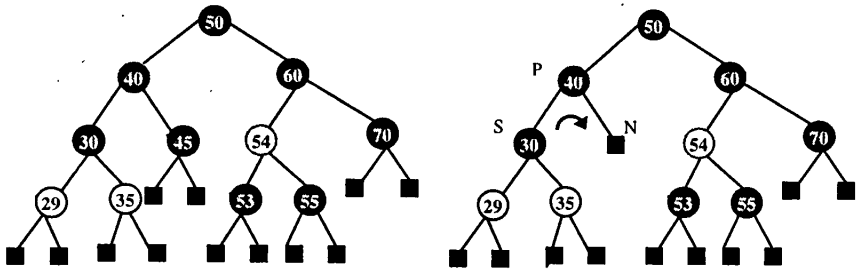
Here the node to be deleted is a black node with a red child, so after deletion the child is painted black.

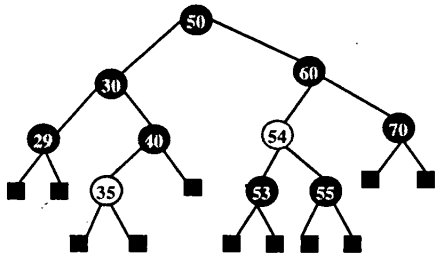
(iii) Delete 70



Node's sibling is red, so case R_1 applies. Perform a right rotation and now sibling is 55 which is black. Both nephews are also black, parent is red and this is case R_2a . Color the sibling red and parent black and we are done.

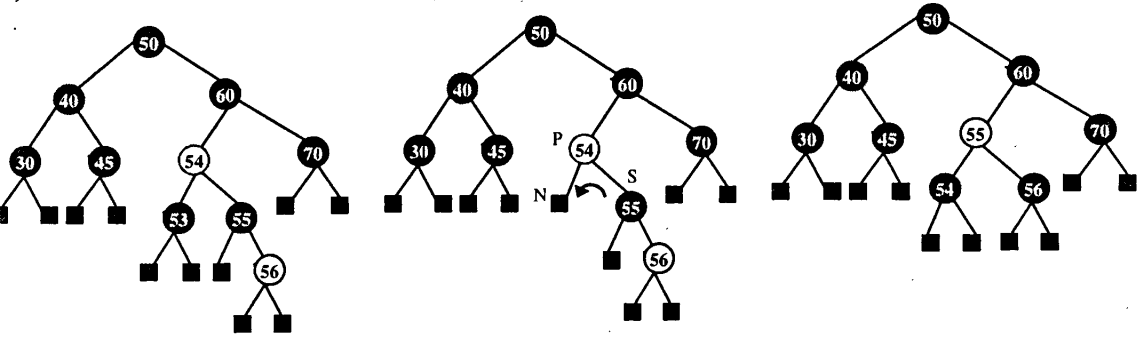
(iv) Delete 45





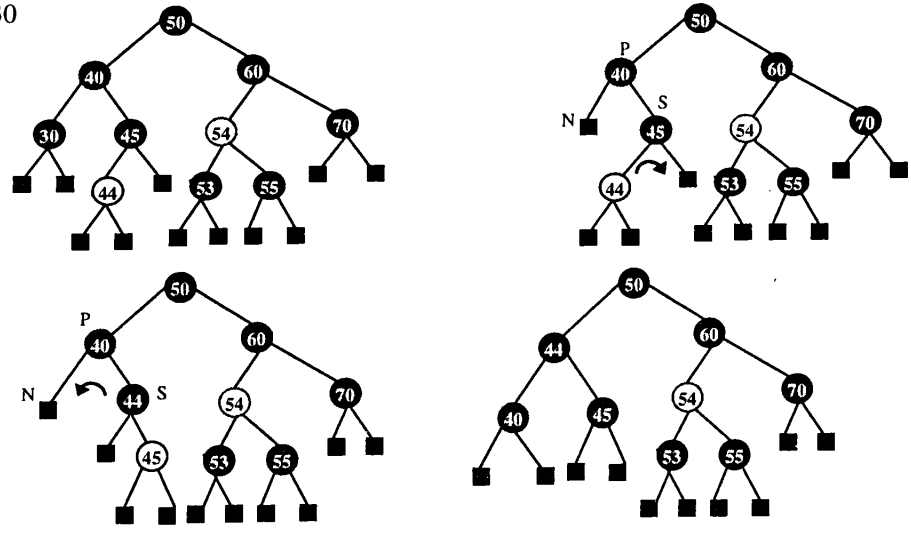
Here N has black sibling. Far nephew is red, so case R_3b applies and a right rotation and recoloring is done.

v) Delete 53



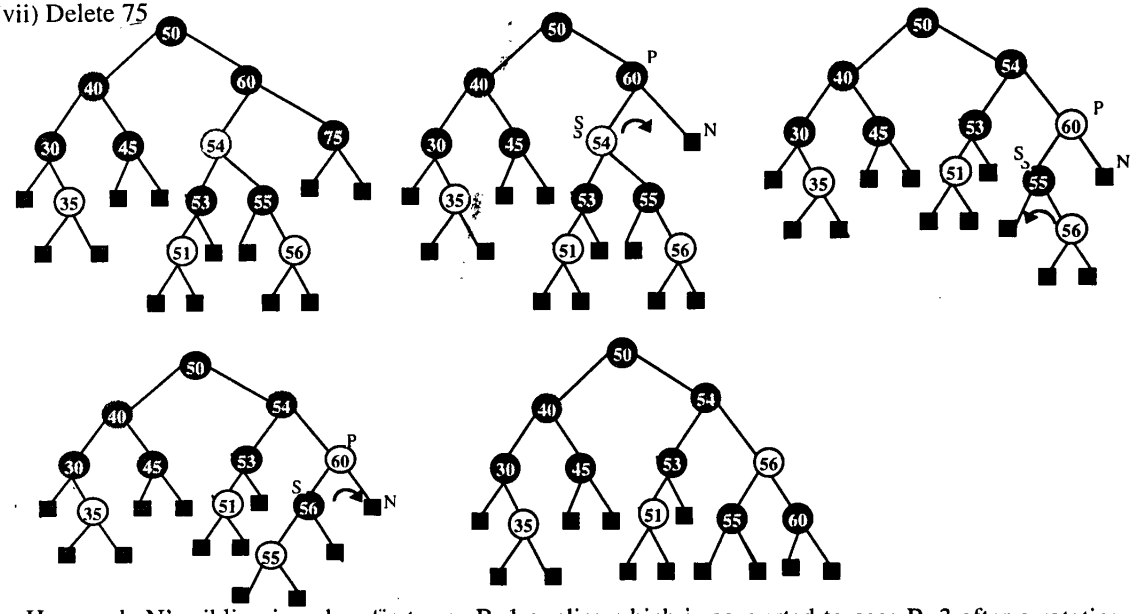
Here N has black sibling and far nephew is red so case L_3b applies and a left rotation and recoloring is done.

vi) Delete 30



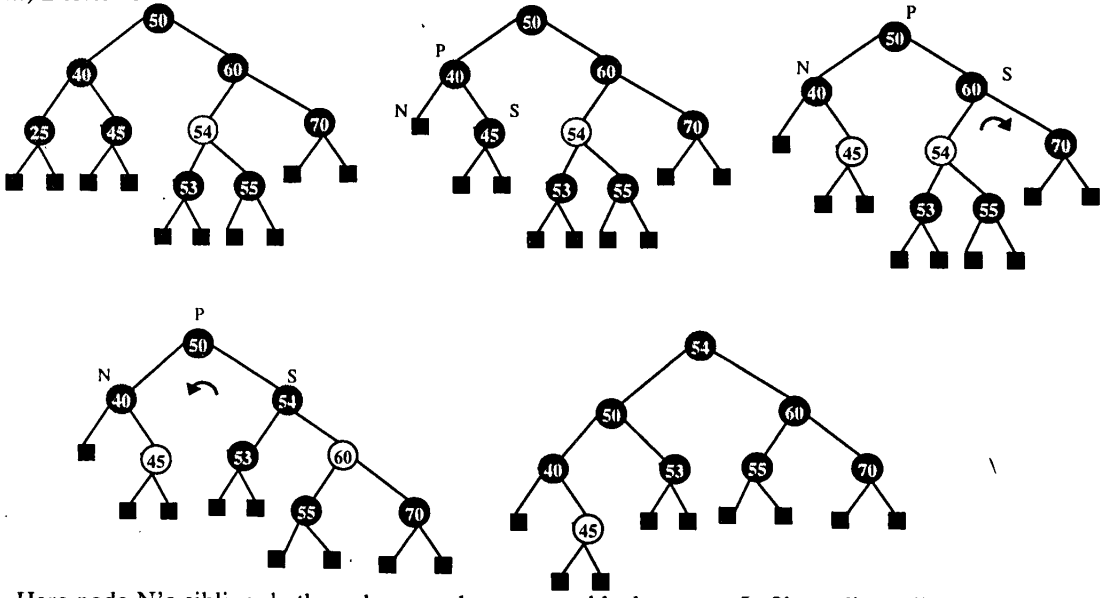
Here node N has black sibling, far nephew is black and near nephew is red, so case L_3a applies which is converted to case L_3b.

(vii) Delete 75



Here node N's sibling is red so first case R_1 applies which is converted to case R_3 after a rotation. After this case R_3a applies which is converted to case R_3b.

(viii) Delete 25



Here node N's sibling, both nephews and parent are black so case L_2b applies. Sibling node 45 is colored red and then case L_3a applies which is converted to case L_3b.

In the implementation, the external nodes are represented by a single sentinel node to save space and this sentinel node also serves as the parent of the root node. The sentinel node is like any other node of the tree and it is colored black while the values of other fields are insignificant.

```
/*P6.7 Program of Red black tree*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        enum {black, red} color;
        int info;
        struct node *lchild;
        struct node *rchild;
        struct node *parent;
};
int find(int item,struct node **loc);
void insert(int);
void insert_balance(struct node *nptr);
void del(int);
void del_balance(struct node *ptr);
void RotateLeft(struct node *ptr);
void RotateRight(struct node *ptr);
struct node *succ(struct node *ptr);
void inorder(struct node *ptr);
void display(struct node *ptr,int level);
struct node *root;
struct node *sentinel;/*will be parent of root node and replace NULL*/
main()
{
        int choice,num;
        sentinel = (struct node *)malloc(sizeof(struct node));
        sentinel->info = -1;
        sentinel->color = black;
        root = sentinel;

        while(1)
        {
                printf("\n");
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Inorder Traversal\n");
                printf("4.Display\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        printf("Enter the number to be inserted : ");
                        scanf("%d",&num);
                        insert(num);
                        break;
                 case 2:
                        printf("Enter the number to be deleted : ");
                        scanf("%d",&num);
                        del(num);
                        break;
                 case 3:
                        inorder(root);
                        break;
                 case 4:
                        display(root,1);
                        break;
                 case 5:
                        exit(1);
                 default:
                        printf("Wrong choice\n");
```

```
                }/*End of switch */
        }/*End of while */
}/*End of main()*/
int find(int item,struct node **loc)
{
        struct node *ptr;
        if(root==sentinel)/*Tree is empty*/
        {
                *loc=sentinel;
                return 0;
        }
        if(item==root->info)/*Item is at root*/
        {
                *loc=root;
                return 1;
        }
        /*Initialize ptr*/
        if(item < root->info)
                ptr=root->lchild;
        else
                ptr=root->rchild;
        while(ptr!=sentinel)
        {
                if(item==ptr->info)
                {
                        *loc=ptr;
                        return 1;
                }
                if(item < ptr->info)
                        ptr=ptr->lchild;
                else
                        ptr=ptr->rchild;
         }//End of while
         *loc=sentinel;          /*Item not found*/
         return 0;
}/*End of find()*/
void insert(int ikey)
{
        struct node *tmp,*ptr,*par;
        par = sentinel;
        ptr = root;
        while(ptr != sentinel)
        {
                par = ptr;
                if(ikey < ptr->info)
                        ptr = ptr->lchild;
                else if(ikey > ptr->info)
                        ptr = ptr->rchild;
                else
                {
                        printf("Duplicate\n");
                        return;
                }
        }
        tmp = (struct node *) malloc(sizeof(struct node));
        tmp->info = ikey;
        tmp->lchild = sentinel;
        tmp->rchild = sentinel;
        tmp->color = red;
        tmp->parent = par;
        if(par==sentinel)
                root = tmp;
        else if(tmp ->info < par->info)
                par->lchild = tmp;
```

```
        else
                par->rchild = tmp;
        insert_balance(tmp);
}/*End of insert()*/
void insert_balance(struct node *nptr)
{
        struct node *uncle,*par,*grandPar;
        while(nptr->parent->color == red)
        {
                par = nptr->parent;
                grandPar = par->parent;
                if(par == grandPar->lchild)
                {
                        uncle = grandPar->rchild;
                        if(uncle->color == red)        /*Case L_1*/
                        {
                                par->color = black;
                                uncle->color = black;
                                grandPar->color =red;
                                nptr = grandPar;
                        }
                        else    /*Uncle is black*/
                        {
                            /           if(nptr == par->rchild) /*Case L_2a*/
                                    {
                                            RotateLeft(par);
                                            nptr = par;
                                            par = nptr->parent;
                                    }
                                    par->color = black;     /*Case L_2b*/
                                    grandPar->color = red;
                                    RotateRight(grandPar);
                        }
                }
                else
                {
                    if(par == grandPar->rchild)
                        {
                                uncle = grandPar->lchild;
                                if(uncle->color == red)  /*Case R_1*/
                                {
                                        par->color = black;
                                        uncle->color = black;
                                        grandPar->color =red;
                                        nptr = grandPar;
                                }
                                else    /*uncle is black*/
                                {
                                        if(nptr == par->lchild)   /*Case R_2a*/
                                        {
                                                RotateRight(par);
                                                nptr = par;
                                                par = nptr->parent;
                                        }
                                        par->color = black;     /*Case R_2b*/
                                        grandPar->color = red;
                                        RotateLeft(grandPar);
                                }
                        }
                }
        }
        root->color = black;
}/*End of insert_balance()*/
```

```
void del(int item)
{
        struct node *child,*ptr,*successor;
        if(!find(item,&ptr))
        {
                printf("Item not present \n");
                return;
        }
        if(ptr->lchild != sentinel || ptr->rchild != sentinel)
        {
                successor = succ(ptr);
                ptr->info = successor->info;
                ptr=successor;
        }
        if(ptr->lchild !=sentinel)
                child = ptr->lchild;
        else
                child = ptr->rchild;
        child->parent = ptr->parent;
        if(ptr->parent == sentinel)
                root=child;
        else if(ptr == ptr->parent->lchild)
                ptr->parent->lchild = child;
        else
                ptr->parent->rchild = child;
        if(child==root)
                child->color = black;
        else if(ptr->color == black)/*black node*/
        {
                if(child != sentinel) /*one child which is red*/
             child->color = black;
                else    /*no child*/
                        del_balance(child);
        }
}/*End of del()*/

void del_balance(struct node *nptr)
{
        struct node *sib;
        while(nptr!=root)
        {
          if(nptr == nptr->parent->lchild)
          {
                sib = nptr->parent->rchild;
                if(sib->color == red)/*Case L_1*/
                {
                        sib->color = black;
                        nptr->parent->color = red;
                        RotateLeft(nptr->parent);
                        sib = nptr->parent->rchild; /*new sibling*/
                }
                if(sib->lchild->color==black && sib->rchild->color==black)
                {
                        sib->color=red;
                        if(nptr->parent->color == red)/*Case L_2a*/
                        {
                                nptr->parent->color = black;
                                return;
                        }
                        else
                            nptr=nptr->parent; /*Case L_2b*/
                }
                else
                {
```

```
                if(sib->rchild->color==black)   /*Case L_3a*/
                {
                        sib->lchild->color=black;
                        sib->color=red;
                        RotateRight(sib);
                        sib = nptr->parent->rchild;
                }
                sib->color = nptr->parent->color;   /*Case L_3b*/
                nptr->parent->color = black;
                sib->rchild->color = black;
                RotateLeft(nptr->parent);
                return;
        }
    }
    else
    {
        sib = nptr->parent->lchild;
        if( sib->color == red )/*Case R_1*/
        {
                sib->color = black;
                nptr->parent->color = red;
                RotateRight(nptr->parent);
                sib = nptr->parent->lchild;
        }
        if(sib->rchild->color==black && sib->lchild->color==black)
        {
                sib->color=red;
                if(nptr->parent->color == red)/*Case R_2a*/
                {
                        nptr->parent->color = black;
                        return;
                }
                else
                        nptr=nptr->parent; /*Case R_2b*/
        }
        else
        {
                if(sib->lchild->color==black)  /*Case R_3a*/
                {
                        sib->rchild->color=black;
                        sib->color=red;
                        RotateLeft(sib);
                        display(root,1);
                        sib = nptr->parent->lchild;
                }
                sib->color = nptr->parent->color;/*case R_3b*/
                nptr->parent->color = black;
                sib->lchild->color = black;
                RotateRight(nptr->parent);
                return;
        }
    }
  }/*End of while*/
/*End of del_balance()*/

void RotateLeft(struct node *pptr)

  struct node *aptr;
  aptr = pptr->rchild;   /*aptr is right child of pptr*/
  pptr->rchild= aptr->lchild;
  if(aptr->lchild !=sentinel)
        aptr->lchild->parent = pptr;
  aptr->parent = pptr->parent;
```

```
        if(pptr->parent == sentinel)
                root = aptr;
        else if(pptr == pptr->parent->lchild)
                pptr->parent->lchild = aptr;
        else
                pptr->parent->rchild = aptr;
        aptr->lchild = pptr;
        pptr->parent = aptr;
}/*End of RoatateLeft()*/

void RotateRight(struct node *pptr)
{
        struct node *aptr;
        aptr = pptr->lchild;
        pptr->lchild= aptr->rchild;
        if(aptr->rchild !=sentinel)
                aptr->rchild->parent = pptr;
        aptr->parent = pptr->parent;
        if(pptr->parent == sentinel)
                root = aptr;
        else if(pptr == pptr->parent->rchild)
                pptr->parent->rchild = aptr;
        else
                pptr->parent->lchild = aptr;
        aptr->rchild = pptr;
        pptr->parent = aptr;
}/*End of RotateRight()*/
struct node *succ(struct node *loc)
{
        struct node *ptr=loc->rchild;
        while(ptr->lchild!=sentinel)
        {
                ptr=ptr->lchild;
        }
        return ptr;
}/*End of succ()*/
void inorder(struct node *ptr)
{
        if(ptr!=sentinel)
        {
                inorder(ptr->lchild);
                printf("%d  ",ptr->info);
                inorder(ptr->rchild);
        }
}/*End of inorder()*/

void display(struct node *ptr,int level)
{
        int i;
        if(ptr!=sentinel)
        {
                display(ptr->rchild,level+1);
                printf("\n");
                for(i=0; i<level; i++)
                        printf("    ");
                printf("%d",ptr->info);
                if(ptr->color==red)
                        printf("^");
                else
                        printf("*");
                display(ptr->lchild,level+1);
        }
}/*End of display()*/
```

6.16 Heap

Heap is a binary tree which satisfies the following two properties-

(i) Structure property – All the levels have maximum number of nodes except possibly the last level. In the last level, all the nodes occur to the left.

(ii) Heap order property – The key value in any node N is greater than or equal to the key values in both its children.

From the structure property, we can see that a heap is a complete binary tree and so its height is $\lceil \log_2(n+1) \rceil$. From the heap order property, we can say that key value in any node N is greater than or equal to the key value in each successor of node N. This means that root node will have the highest key value.

The heap that we have defined above is called max heap or descending heap. Similarly we can define a min heap or ascending heap. If in the second property, we change "greater" to "smaller" then we get a min heap. In a min heap, the root will have the smallest key value. The trees in figure 6.127 are max heaps and the trees in figure 6.128 are min heaps.

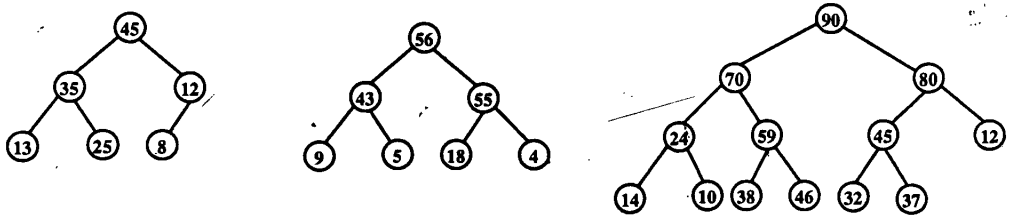


Figure 6.127 Max heaps

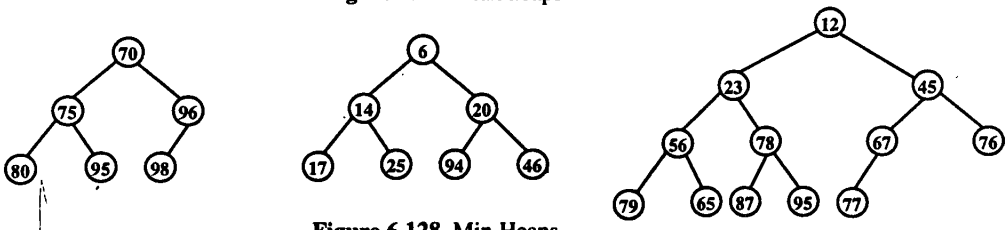


Figure 6.128 Min Heaps

The nodes in a heap are partially ordered i.e. there is no ordering of left and right child, any of the two children can be greater than the other. So a heap is not a search tree.

Heap can be used in problems where the largest(or smallest) value has to be determined quickly. The largest or smallest value can also be determined by sorting the data in descending or ascending order and picking up the first element. But in that case we have to do more work than required because we just need to find the largest(or smallest) element and for that we have to keep all the elements in order. From now onwards in our discussion we will use max heap only.

Heap is a complete binary tree and we know that sequential representation is efficient for these types of trees so heaps are implemented using arrays. Another advantage in sequential representation is that we can easily move up and down the tree, while in linked representation we would have to maintain an extra pointer in each node to move up the tree.

If the root is stored at index 1 of the array, then left and right child of any node N located at index i will be located at indices 2i and 2i+1 and parent of node N will be at index floor(i/2).

While implementing heap in an array, we will maintain a variable which will represent the size of the heap i.e. number of nodes currently in the heap. If n is the heap size then nodes of heap are stored in `arr[1]`, `arr[2].....arr[n]`. The elements of array after `arr[n]` may contain valid entries but they are not part of the heap.

If the value of 2i or 2i+1 is greater than the heap size, then the corresponding child does not exist.The following figure shows the array representation of a heap of size 12.

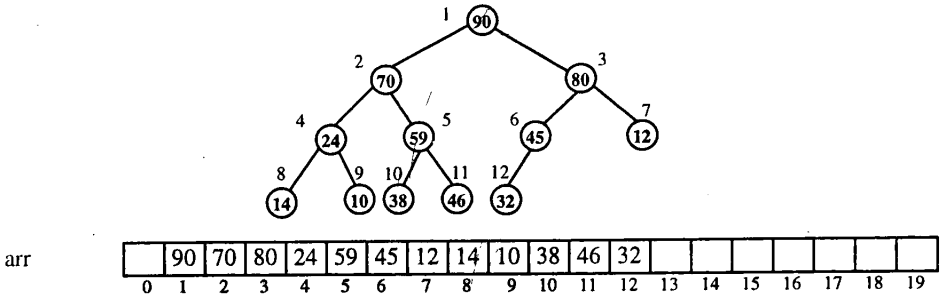


Figure 6.129

Node 45 does not have a right child because 2*6+1=13 is greater than heap size which is 12. In location 0 of the array we can store a sentinel value.

Now we will see how to insert and delete elements from a heap, but before that let us have a look at the `main()` function and other declarations needed for our program of heap.

```
/*P6.8 Program for insertion and deletion in heap*/
#include <stdio.h>
#define MAX_VAL 9999 /*All values in heap must be less than this value*/
void insert(int num, int arr[],int *p_hsize);
int del_root(int arr[],int *p_hsize);
void restoreUp(int arr[],int loc);
void restoreDown(int arr[],int i, int size);
void buildHeap(int arr[],int size );
main()
{
        int arr[100]; /*array used to represent heap*/
        int hsize=0;  /*Number of nodes in the heap*/
        int i,choice,num;
        arr[0]= MAX_VAL;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Delete root\n");
                printf("3.Display\n");
                printf("4.Build Heap\n");
                printf("5.Exit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1:
                        printf("Enter the number to be inserted : ");
                        scanf("%d",&num);
                        insert(num,arr,&hsize);
                        break;
                 case 2:
                        if(hsize==0)
                                printf("Heap is empty \n");
                        else
                        {
                                num = del_root(arr,&hsize);
                                printf("Maximum element is %d\n", num);
                        }
                        break;
                 case 3:
                        display(arr,hsize);
                        break;
                 case 4:
```

```
                printf("Enter size of the array ");
                scanf("%d",&hsize);
                printf("Enter array : ");
                for(i=1; i<=hsize; i++)
                        scanf("%d",&arr[i]);
                buildHeap(arr,hsize);
                break;
        case 5:
                exit(1);
        default:
                printf("Wrong choice\n");
        }/*End of switch */
    }/*End of while */
}/*End of main()*/
```

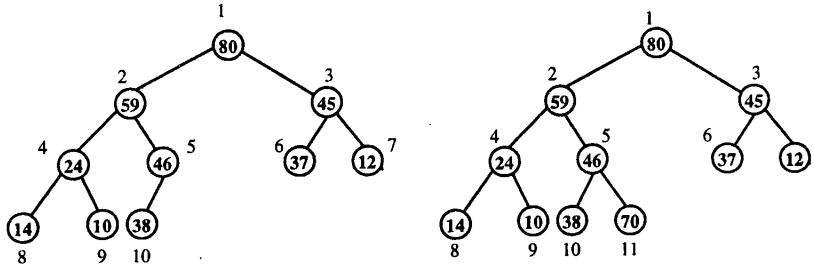
We have taken an array named `arr` which will be used to represent the heap. The variable `hsize` is used to denote the number of elements in the heap and it is initialized to zero. We have defined a symbolic constant MAX_VAL which is stored in the 0^{th} index of the array and it serves as the sentinel value. All the values in the heap should be less than this constant.

The function `display()` can be written as-

```
void display(int arr[],int hsize)
{
    int i;
    if(hsize==0)
    {
        printf("Heap is empty\n");
        return;
    }
    for(i=1; i<=hsize; i++)
        printf("%d ",arr[i]);
    printf("\n");
    printf("Number of elements = %d\n",hsize);
}/*End of display()*/
```

6.16.1 Insertion in Heap

We should insert the new key in such a way that both properties hold true after insertion also. If n is the size of heap before insertion then it is increased to n+1 and the new key is inserted in `arr[n+1]`. This location is the next leaf node in the heap and thus the heap remains a complete binary tree after insertion also. Suppose we have to insert key 70 in the heap tree given below-



Heap size is 10
Key 70 to be inserted in this heap

Heap size increased to 11
Key 70 inserted in arr[11]

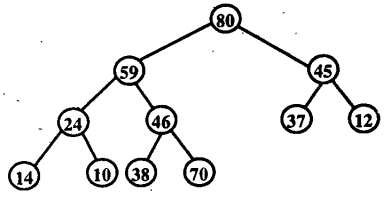
Figure 6.130

Insertion in this way ensures that the resulting tree fulfills the structure property i.e. tree remains a complete binary tree but the second property may be violated. For example in this case we can see that 70 is greater than its parent 46, which is a violation of the heap order property. To restore the heap order property we perform a

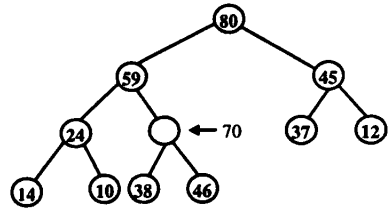
procedure called restoreUp in which we will move the newly inserted key up the heap and place it in its appropriate place.

Suppose k is the key that violates the heap order property and needs to be moved up. Compare k with the key in parent node, if parent key is smaller than k, then the parent key is moved down. Now we try to insert k in parent's place and for this k is compared with the key of new parent. This procedure stops when we get a parent key which is greater than k or when we reach the root node. This way we place the key k at its proper place in the heap.

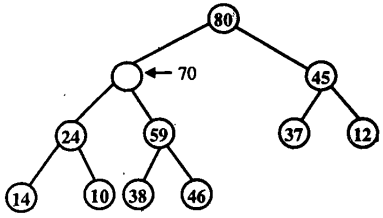
We had inserted the key 70 in the last leaf node of the heap, and it violated the heap order property. Now let us see how we can place the key 70 in the appropriate place using the procedure restoreUp.

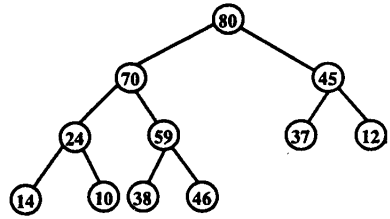


Compare 70 with parent 46
46 is less than 70, move 46 down



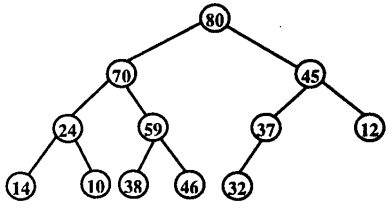
Compare 70 with new parent 59
59 is less than 70, move 59 down



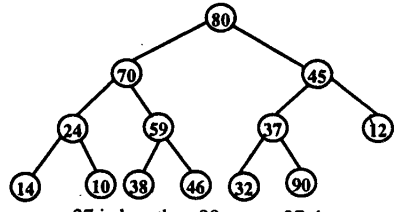


Let us see two more examples of insertion in heap

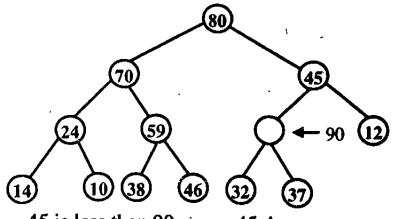
Insert 90



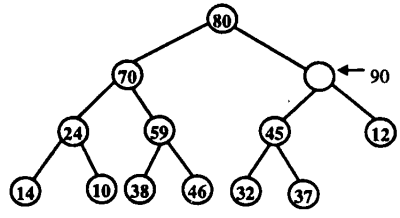
Insert 90 in this heap tree



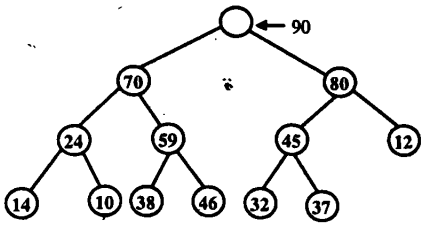
37 is less than 90, move 37 down



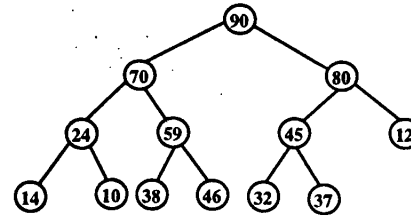
45 is less than 90, move 45 down



80 is less than 90, move 80 down

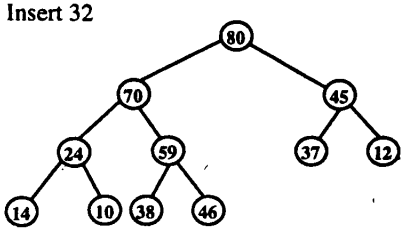


Reached the root node

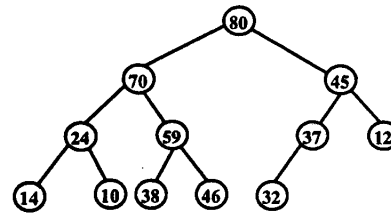


90 placed in root node

Insert 32



Insert 32 in this heap tree



37 is greater than 32, hence 32 is at proper place

We can write the `insert()` function as-

```
void insert(int num,int arr[],int *p_hsize)
{
        (*p_hsize)++;  /*Increase the heap size by 1*/
        arr[*p_hsize]=num;
        restoreUp(arr,*p_hsize);
}/*End of insert()*/
```

Here the first argument is the key to be inserted, second argument is the array which represents the heap and third argument is a pointer to the variable `hsize` of `main()`. Initially the heap size is increased and then the key is inserted at the last position and the function `restoreUp()` is called. The function `restoreUp()` can be written as-

```
void restoreUp(int arr[],int i)
{
        int k = arr[i];
        int par = i/2;

        while(arr[par] < k)
        {
                arr[i]=arr[par];
                i = par;
                par = i/2;
        }
        arr[i] = k;
}/*End of restoreUp()*/
```

We come out of the while loop when exact place for the key is found. Suppose the key is the largest and has to be placed in the root node, value of `i` and `par` will be 1 and 0 respectively. In this case also the while loop terminates because we have stored a very large sentinel value in the location 0 of the array. If we don't store this value in `arr[0]` then we have to add one more condition in the while loop.

```
while(par>=1 && arr[par]<k)
```

So the use of sentinel avoids checking of one condition per loop iteration. Now we will take some keys and insert them in an initially empty heap tree.

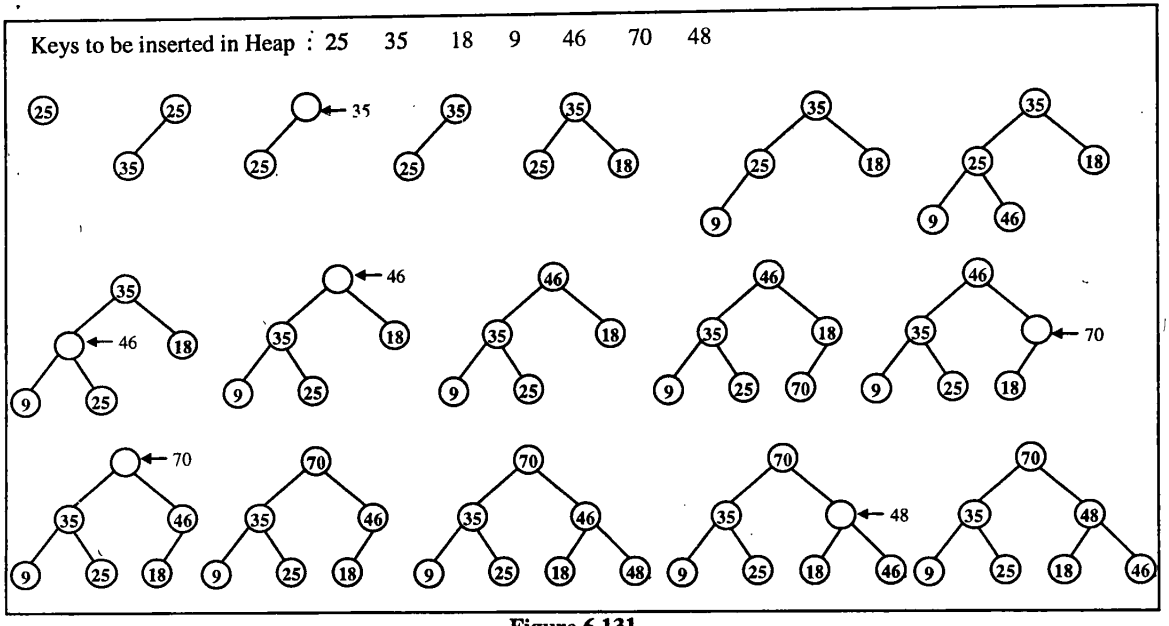


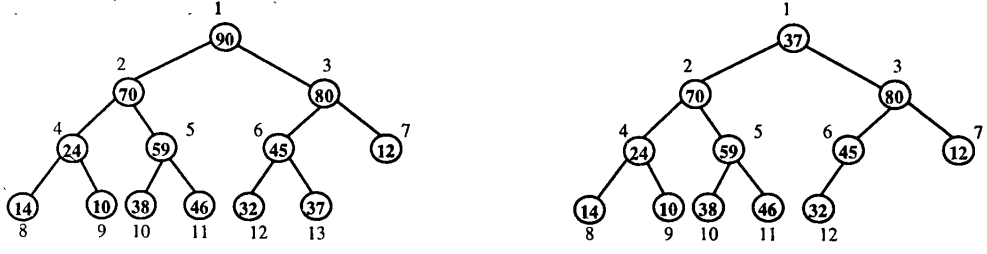
Figure 6.131

For insertion we move on a single path from leaf node towards the root, hence complexity is O(h). Height of heap is $\lceil \log_2(n+1) \rceil$ so complexity is O(log n). The best case of insertion is when the key can be inserted at the last position i.e. there is no need to move it up. The worst case is when the key to be inserted is maximum; it has to be inserted in the root.

6.16.2 Deletion

Any node can be deleted from the heap but deletion of root node is meaningful because it contains the maximum value.

Suppose we have to delete root node from a heap of size n. The key in the root can be assigned to some variable so that it can be processed. Then we copy the key in the last leaf node i.e. key in location `arr[n]` to the root node. After this size of heap is decreased to n-1, hence the last leaf node is deleted from the heap. Let us see how we delete root from the heap in the given figure.



Heap size is 13
Copy key from arr[13] to root node

Heap size reduced to 12

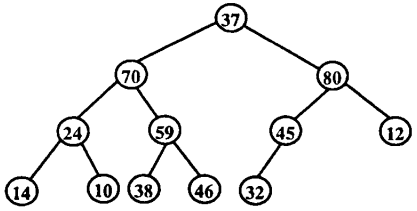
Figure 6.132

The resulting structure after deleting the root in this way is a complete binary tree so the first property still holds true but the heap order property may be violated if key in the root node is smaller than any of its child. In the above example, the key 37 is not at proper place because it is smaller than both of its children. The procedure restoreDown will move the key down and place it at its proper place.

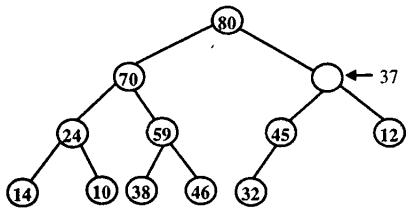
Suppose the key k violates the heap order property. Compare k with both left and right child. If both children are smaller than k, then we are done. If one child is greater than k, then move this greater child up. If both lef

and right child are greater than k, then move the larger of the two children up. After moving the child up, we try to insert the key k in this child's place and for this we compare it with its new children. The procedure stops when both children of k are smaller than k, or when we reach a leaf node.

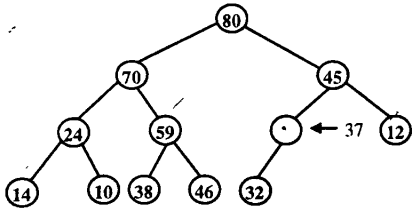
Now let us see how we can place the key 37 in the appropriate place using the procedure restoreDown.



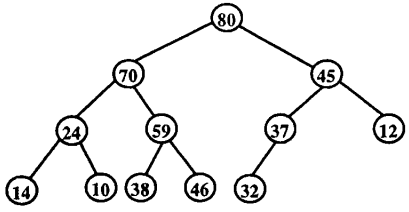
Compare 37 with 70 and 80
Both 70 and 80 greater than 37
80 is larger of two, so move it up



Compare 37 with 45 and 12
45 is greater than 37 so move 45 up

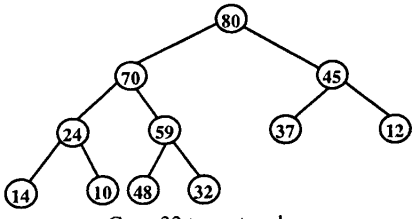


Compare 37 with 32
32 is less than 37, proper place for 37 found

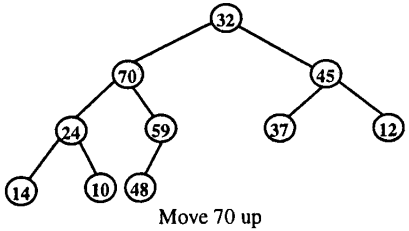


37 placed in appropriate place

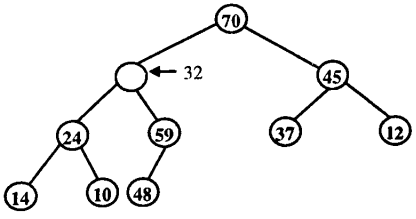
Let us see one more example of deletion of root from the heap.



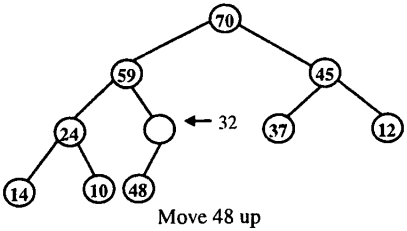
Copy 32 to root node



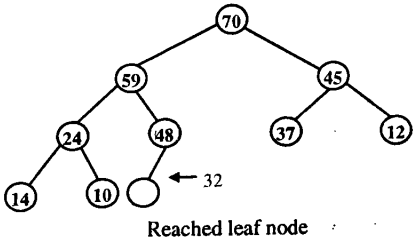
Move 70 up



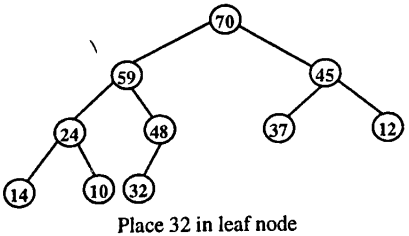
Move 59 up



Move 48 up



Reached leaf node



Place 32 in leaf node

The functions `del_root()` and `restoreDown()` are-

```
int del_root(int arr[],int *p_hsize)
{
        int max = arr[1];          /*Save the element present at the root*/
        arr[1] = arr[*p_hsize]; /*Place the last element in the root*/
        (*p_hsize)--;  /*Decrease the heap size by 1*/
        restoreDown(arr,1,*p_hsize);
        return max;
}/*End of del_root()*/

void restoreDown(int arr[],int i,int hsize)
{
        int lchild=2*i,rchild=lchild+1;
        int num=arr[i];

        while(rchild <= hsize)
        {
                if(num>=arr[lchild] && num>=arr[rchild])
                {
                        arr[i] = num;
                        return;
                }
                else if(arr[lchild] > arr[rchild])
                {
                        arr[i] = arr[lchild];
                        i = lchild;
                }
                else
                {
                        arr[i] = arr[rchild];
                        i = rchild;
                }
                lchild = 2 * i;
                rchild = lchild + 1;
        }
        /*If number of nodes is even*/
        if(lchild==hsize && num<arr[lchild])
        {
                arr[i]=arr[lchild];
                i = lchild;
        }
        arr[i]=num;
}/*End of restoreDown()*/
```

When the number of nodes in heap is odd, all nodes have either 2 children or are leaf nodes and when the number of nodes is even then there is one node that has only left child. For example if the number of nodes is 10 then node at index 5 has only left child which is at index 10. In the function `restoreDown()`, we check for this condition separately.

For deletion we move on a single path from root node towards the leaf, hence complexity is O(h). Height of heap is $\lceil \log_2(n+1) \rceil$, so complexity is O(log n).

6.16.3 Building a heap

If we have to form a heap from some n elements, we can start with an empty heap and insert the elements sequentially n times. The `insert()` function will be called n times to build a heap in this way.

Now let us see how we can form a heap from an array i.e. suppose we have an array of size n in which data is in random order and we have to convert it to a heap of size n.

We start with a heap of size 1, i.e. initially only `arr[1]` is in the heap and after that we insert all the remaining elements of the array(`arr[2]`..... `arr[n]`) in the heap one by one. We insert `arr[2]` in heap of size

1, then `arr[3]` in heap of size 2, and so on till `arr[n]` is inserted in heap of size n-1 and finally we get a heap of size n.

Since the elements are already in an array, there is no need to call `insert()`, we just increase the size of heap and call `restoreUp()` for next element of array.

```
void buildHeap(int arr[],int size)
{
        int i;
        for(i=2; i<=size; i++)
                restoreUp(arr,i);
}/*End of buildHeap()*/
```

This method of building a heap by inserting elements one by one is top down approach.
The worst case is when data is in ascending order i.e. each new key has to rise up to the root, so worst case time for this approach is proportional to O(n log n). We can build heap in O(n) time by using a bottom up approach.

In this method the array is considered to be representing a complete binary tree. Suppose we have an array of 11 elements stored in `arr[1].....arr[11]` and we want to convert this array into a heap.

arr

25	35	18	9	46	70	48	23	78	12	95
1	2	3	4	5	6	7	8	9	10	11

We can think of this array representing a complete binary tree as shown in figure 6.133.

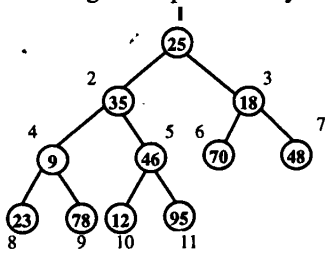


Figure 6.133

Now this structure satisfies the first property. To convert it to a heap we have to make sure that the second property is also satisfied. For this we start from the first non leaf node and call `restoreDown()` for each node of the heap till the root node.

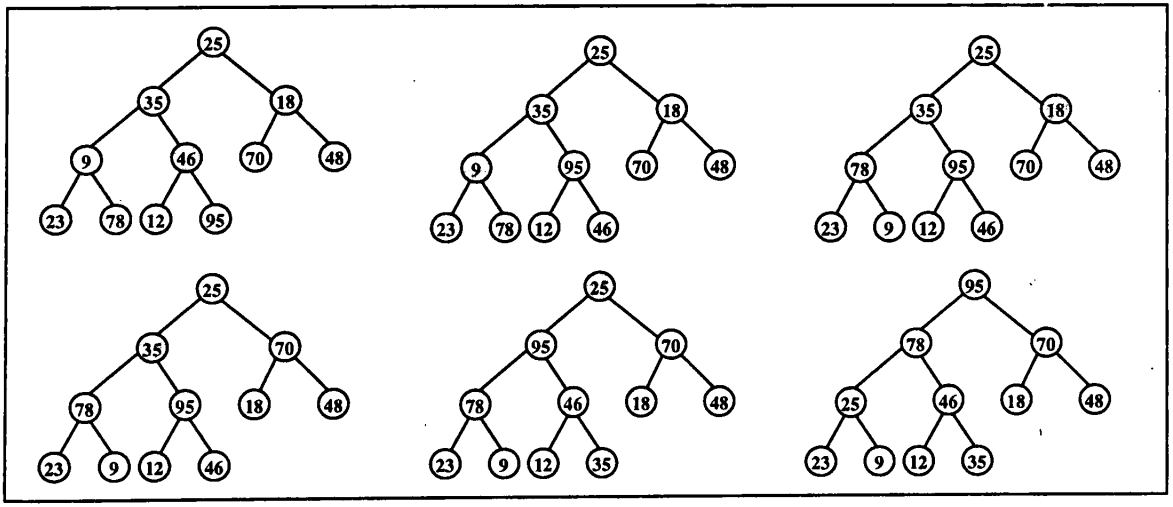


Figure 6.134

The first non leaf node is always present at index floor(n/2) of the array(if root stored at index 1 and heap size is n). So the function `restoreDown()` is called for all the nodes with indices floor(n/2) , floor(n/2)-1, floor(n/2)-2 .....2,1. In our example the heap size is 11, hence the first non leaf node is 46 present at index 5. So in this case `restoreDown()` is called for `arr[5], arr[4], arr[3], arr[2], arr[1]`.

First restoreDown is called for 46. The right child of 46 is 95 which is greater than 46, so 95 is moved up and 46 is moved down. Similarly retsoreDown is called for 9, 18, 35 and 25 and all of them are placed at proper places. Finally we get a heap of size 11. The function `buildHeap()` using bottom up approach is-

```
void buildHeap(int arr[],int size)
{
      int i;
      for(i=size/2; i>=1; i--)
            restoreDown(arr,i,size);
}/*End of buildHeap()*/
```

This was the procedure of building a heap through bottom up approach; now let us understand how we are able to get a heap by applying this procedure.

The heap is constructed by making smaller subheaps from bottom up. Each leaf node is a heap of size 1, so we start working from the first non leaf node. Since we are working from bottom to up, whenever we analyze a node, its left and right subtrees will be heaps. Starting from the first non leaf node, each node N is considered as the root of a subtree whose left and right subtrees are heaps and `restoreDown()` is called for that node so that the whole subtree rooted at node N also becomes a heap.

There are three main applications of heap-

1. Selection algorithm
2. Implementation of priority queue
3. Heap sort

6.16.4 Selection algorithm

There are two ways to find the k^{th} largest element in a list. The first method is to sort the whole list in descending order and the element at k^{th} location will be the k^{th} largest element. The second approach uses a heap and is more efficient, first a heap is built from the elements and then k elements are deleted from the heap. The last element deleted is the k^{th} largest element.

6,16.5 Implementation of Priority Queue

We have already studied about priority queue. If we implement it through a queue then deletion is O(n) while insertion is O(1), and if we implement it through a sorted list then insertion is O(n) while deletion is O(1). If a heap is used to implement a priority queue, then both these operations can be performed in O(log n) time. A max or min heap can be used to implement a max or min priority queue. If we make a max heap from the priorities of nodes, then the node with highest priority will always be at the root node. The find operation in priority queue finds the node with highest priority and delete operation deletes the node with highest priority.

The third application heap sort will be discussed in the chapter of Sorting.

6.17 Weighted path length

Suppose every external node in an extended binary tree is assigned some nonnegative weight, then the external weighted path length of the tree can be calculated as -

$P = W_1P_1 + W_2P_2 + \ldots\ldots\ldots\ldots + W_nP_n$

where W_i denotes the weight and P_i denotes the path length of an external node.

If we create different trees that have same weights on external nodes then it is not necessary that they have same external weighted path length.

Let us take the weights 4, 7, 8, 12 and create three different trees.

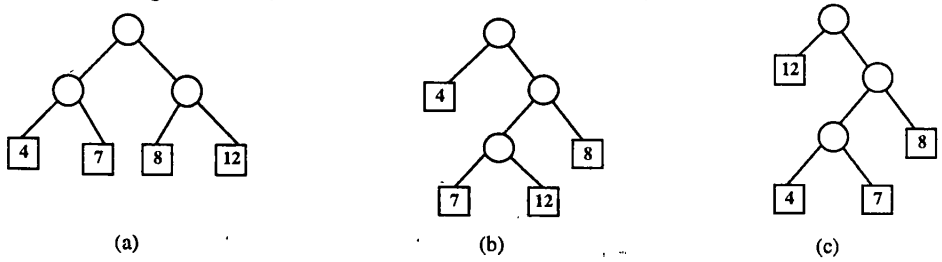


Figure 6.135

The weighted path lengths of these trees is-

Weighted Path Length of (a) : $P_1 = 4*2 + 7*2 + 8*2 + 12*2 = 8 + 14 + 16 + 24 = 62$
Weighted Path Length of (b) : $P_2 = 4*1 + 7*3 + 12*3 + 8*2 = 4 + 21 + 36 + 16 = 77$
Weighted Path Length of (c) : $P_3 = 12*1 + 4*3 + 7*3 + 8*2 = 12 + 12 + 21 + 16 = 61$

The three different trees have different external weighted path lengths. Now our aim is to obtain an extended binary tree which has minimum weighted external path length for n external nodes with weights w_1, w_2, w_3, ...w_n . This tree can be created by Huffman algorithm and is named Huffman tree in the honor of its inventor David Huffman.

6.18 Huffman Tree

Huffman tree is built from bottom up rather than top down i.e. the creation of trees starts from leaf nodes and proceeds upwards.

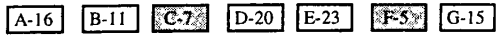
Suppose we have n elements with weights w_1, w_2.....w_n and we want to construct a Huffman tree for these set of weights.

For each element we create a tree with a single root node. So initially we have a forest of n trees, each data item with its weight is placed in its own tree.

In each step of the algorithm, we pick up two trees T_i and T_j with smallest weights(w_i and w_j) and combine them into a new tree T_k. The two trees T_i and T_j become subtrees of this new tree T_k and the weight of this new tree is the sum of the weights of the trees T_i and T_j i.e. $w_i + w_j$. After each step, the number of trees in our forest will decrease by 1. The process is continued till we get a single tree and this is the final Huffman tree. The leaf nodes of this tree contain the elements and their weights. Let us take 7 elements with weights and create an extended binary tree by Huffman algorithm.

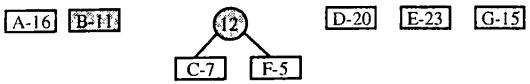
	A	B	C	D	E	F	G
Weight	16	11	7	20	23	5	15

(i) Initially we have a forest of 7 single node trees.



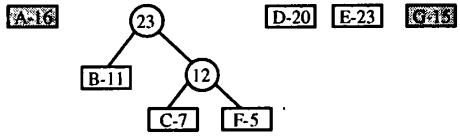
The two trees with smallest weights are trees weighted 7 and 5(shaded in the figure).

(ii) The trees weighted 7 and 5 are combined to form a new tree weighted 12.



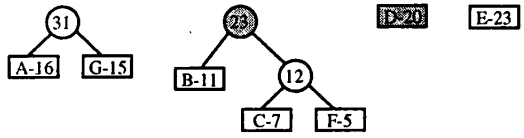
We have made tree weighted 7 as the left child arbitrarily, any tree can be made left or right child. Now the two trees with smallest weights are trees weighted 11 and 12.

(iii) The trees weighted 11 and 12 are combined to form a new tree weighted 23.



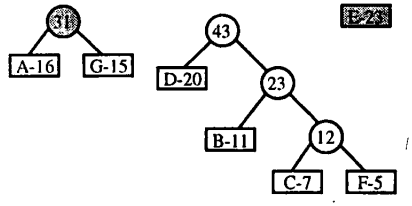
Now the two trees with smallest weights are trees weighted 16 and 15.

(iv) The trees weighted 16 and 15 are combined to form a new tree weighted 31.



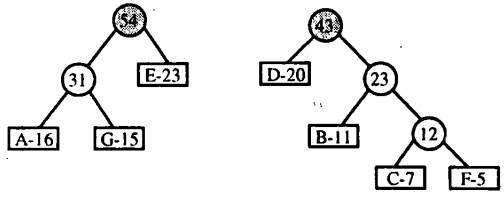
Now the two smallest weights are 20 and 23, but there are two trees with weight 23. We can break this tie arbitrarily and chose any one of them for combining.

(v) The trees weighted 20 and 23 are combined to form a new tree weighted 43.



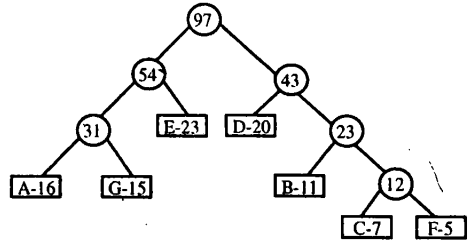
Now the two trees with smallest weights are trees weighted 31 and 23.

(vi) The trees weighted 31 and 23 are combined to form a new tree weighted 54.



Now the two trees with smallest weights are trees weighted 54 and 43.

(vii) The trees weighted 54 and 43 are combined to form a new tree weighted 97.



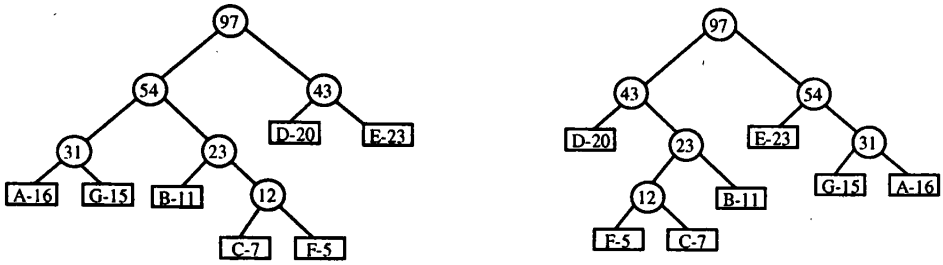
Now only a single tree is left and this is the final Huffman tree. The weighted path length for this tree is-

16*3 + 15*3 + 23*2 + 20*2 + 11*3 + 7*4 + 5*4

=260

In the above procedure, we have seen that when two trees are combined anyone can be made left or right subtree of the new tree and if there is more than one tree with equal weights in the roots, then we can arbitrarily chose anyone for combining. So the Huffman tree produced by Huffman algorithm is not unique. There can be

different Huffman trees for same set of weights, but the weighted path length for all of them would be same irrespective of the shape of the tree. The following figure shows two more Huffman trees for the same set of weights given in the example.



The weighted path lengths for these two trees are-
16*3 +15*3 + 11*3 + 7*4 + 5*4 + 20*2 +23*2 = 260
20*2 + 5*4 + 7*4 + 11*3 + 23*2 + 15*3 + 16*3 = 260
This algorithm uses greedy approach because at each step we choose the two trees that appear to be the best candidates for combining.

6.18.1 Application of Huffman tree : Huffman Codes

Suppose we have a long message that comprises of different symbols. We want to encode this message with the help of bits(0s and 1s). To do this, we will assign a binary code to each symbol and then concatenate these individual codes to get the code for the message. If we have n distinct symbols, then we can encode each symbol by using a r-bit code where $2^{r-1} < n \leq 2^r$.
Suppose we have 5 different symbols v, w, x, y, z. We will need a 3-bit code to uniquely represent each symbol.
$2^2 < 5 \leq 2^3$

Suppose the codes assigned to these symbols are-

v	w	x	y	z
000	001	010	011	100

If we use the above codes then the message "wyxwvyz" can be encoded as

00101101000010000011100

This type of coding is called fixed length coding because the length of code for each symbol is same. ASCII and EBCDIC are fixed length codes. If we know the frequency(number of occurrences) of symbols in a message, we can reduce the size of coded message using variable length codes. Instead of assigning equal bits to all the symbols, we can give shorter codes to symbols that occur more frequently and longer codes to symbols that occur less frequently. These types of codes are called variable length codes because here the length of code of each symbol is different. This way we can achieve data compression. The following tables show the comparison of fixed length codes and variable length codes.

Fixed length code			
Character	Frequency	Code	Bits
v	5	000	5*3=15
w	36	001	36*3=108
x	25	010	25*3=75
y	58	011	58*3=174
z	8	100	8*3=24
Total Bits			**396**

Variable length code			
Character	Frequency	Code	Bits
v	5	0100	5*4=20
w	36	00	36*2=72
x	25	011	25*3=75
y	58	1	58*1=58
z	8	0101	8*4=32
Total Bits			**257**

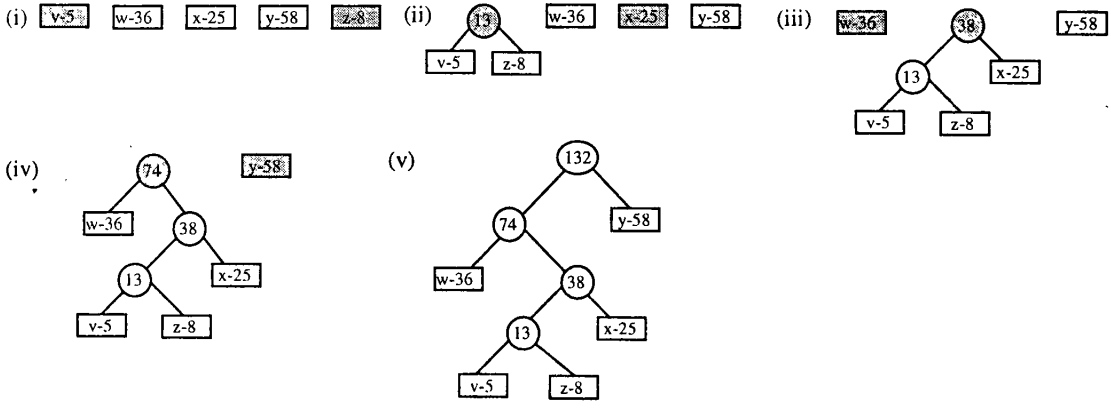
In fixed length code, all the symbols are assigned 3-bit codes irrespective of their frequency. The coded message occupies 396 bits in this case. In variable length codes, we have assigned 1 bit code to y since it is most frequent while v and z are assigned 4-bit codes since they are less frequent. So the same message can be coded using only 257 bits if variable length codes are used.

Encoding a message using variable length codes is simple, and as in fixed length codes here also the individual codes of symbols are just concatenated to get the code for the message. For example if we use the variable length codes given in the table above, then the message "wyxwvyz" would be encoded as 00101100010010101.

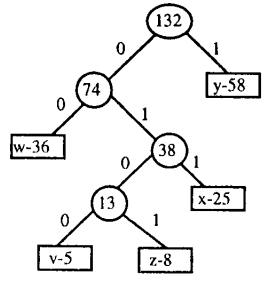
Now let us see how we can decode the encoded message. When fixed length codes are used decoding is simple – start reading bits from the left and just transform each 3 bit sequence to the corresponding symbol. When variable length codes are used we don't know how many bits to pick. For example if the encoded message is 0010101011, the code for first symbol may be 0 or 00 or 001 or 0010. From the table we can see that 0 is not a code, but 00 is a code for w and no other code starts with 00. So the first two bits can be decoded to get the symbol w. The next bit is 1, which is code of y, and no other code starts with 1 so the next symbol decoded is y. Next bit is 0 which is not a code, 01 is also not a code, 010 is also not a code, 0101 is code of z. So next decoded symbol is z.

This decoding method works because the variable length codes that we have made are prefix free i.e. no code is a prefix of other code. For example 00 is the code for symbol w, so it is not prefix of any other code i.e. no code starts with 00. Similarly no code starts with 1 since it is code of y. These types of codes are known as prefix codes. Now the question is how we can obtain these variable length codes that are prefix free.

Huffman tree is used to generate these codes and the resulting codes are called Huffman codes. First of all we will create a Huffman tree for the data given in the table. Each external node contains a symbol and its weight is equal to the frequency of the symbol.



Now let us see how this Huffman tree can be used to generate Huffman codes. Each branch in the Huffman tree is assigned a bit value. The left branches are assigned 0 and right branches are assigned 1.



We can find out the code of each symbol by tracing the path that starts from the root and ends at the leaf node that contains the given symbol. All the bit values in this path are recorded and the bit sequence so obtained is the code for the symbol. For example to get the code of v, first we move left(0), then right (1) then left(0) and then again left(0). So the code for v is 0100. Similarly we can find the codes of other symbols which are-

w	y	x	v	z
00	1	011	0100	0101

Since all the symbols are in the leaf nodes, this method generates codes that are prefix free and hence decoding will be unambiguous.

The same Huffman tree is used for decoding data. The encoded data is read bit by bit from the left side. We will start from the root, if there is a 0 in the coded message we go the left child and if there is 1 in the coded message we go the right child and this procedure continues till we reach the leaf node. On reaching the leaf node, we get the symbol and then again we start from the root node. Let us take a coded message and decode it.

0001000110101001011

00 01000110101001011	w
00 0100 0110101001011	wv
00 0100 011 0101001011	wvx
00 0100 011 0101 001011	wvxz
00 0100 011 0101 00 1011	wvxzw
00 0100 011 0101 00 1 011	wvxzwy
00 0100 011 0101 00 1 011	wvxzwyx

The drawback with Huffman codes is that we have to scan the data twice – first time for getting the frequencies and next time for actual encoding.

6.19 General Tree

We had looked at the definition of a general tree in the beginning of this chapter. The following are the three differences in binary tree and general tree-

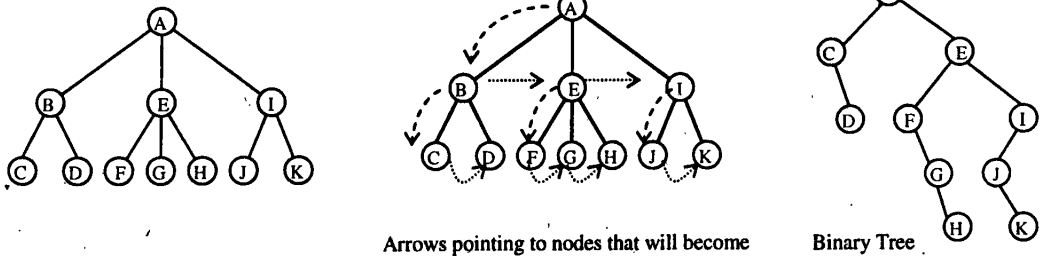
1) A binary tree may be empty, but there should be at least one node in a general tree.

2) In a binary tree, each node can have at most two children but in a general tree a node may have more than two children.

3) In a binary tree, every child is left or right child but in a general tree a child cannot be distinguished as left or right. All the children of a node in a general tree are termed as siblings of each other.

A general tree can be converted to binary tree as -

Root of binary tree will be same as that of general tree. First child of a node in general tree will be the left child of that node in binary tree. The next sibling of a node in general tree will be the right child of that node in binary tree. Let us take two examples of general tree and create corresponding binary trees.



General Tree

Arrows pointing to nodes that will become left and right children in binary tree

Binary Tree

Figure 6.136

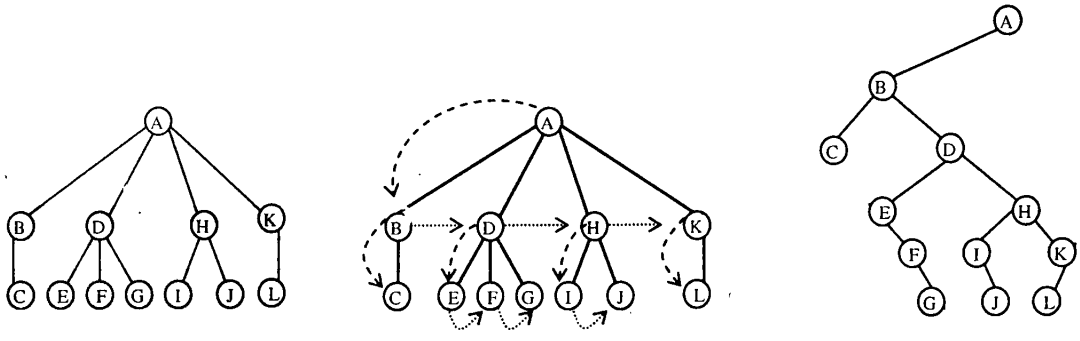


Figure 6.137

In both the examples, the first figure shows the general tree, the second figure shows the same tree with arrows representing left and right child of a node in binary tree and the third figure shows the corresponding binary tree. Note that in any general tree the root does not have a sibling, so the root of the corresponding binary tree will not have any right child.

A general tree can easily be represented using the binary tree format. The structure for a node of general tree can be taken as-

```
struct node
{
        int info;
        struct node *firstChild;
        struct node *nextSibling;
};
```

In a binary tree, left and right pointers of a node pointed to the left and right child of the node respectively. In general tree, left pointer will point to the first child or the leftmost child of the node and right pointer will point to the next sibling of the node. The linked representation of a general tree is given next-

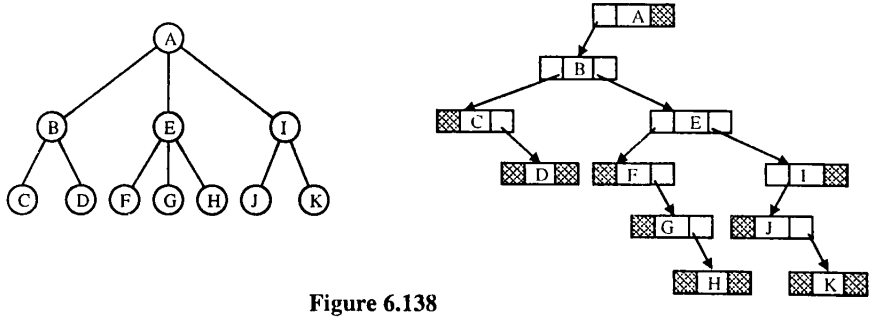


Figure 6.138

6.20 Multiway search tree

Till now we have studied about internal searching only, i.e. we have assumed that data to be searched is present in primary storage area. Now we will study about external searching in which data is to be retrieved from secondary storage like disk files. We know that the access time in the case of secondary storage is much more than that of primary storage. So while doing external searching we should try to reduce the number of accesses. When data is accessed from a disk, a whole block is read instead of a single word. Keeping these points in mind a data structure was devised which was specially suited for external searching and is known as multiway search tree. Till now in all the trees that we have studied, a node can hold only one key value and can have at most two children. In a multiway search tree a node can hold more than one value and can have more than 2 children.

Usually the size of the node is made to coincide with the size of the block of the secondary storage device. Due to the large branching factor of multiway search trees their height is reduced, so the number of nodes traversed to reach a particular record also decreases which is desirable in external searching.

A multiway search tree of order m is a search tree in which any node can have at the most m children. The properties of a non empty m way search tree of order m are-

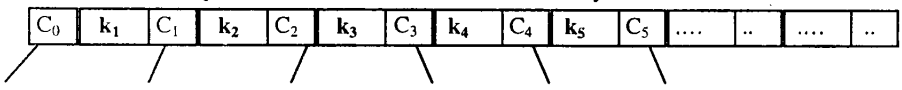
(i) Each node can hold maximum m-1 keys and can have maximum m children.

(ii) A node with n children has n-1 key values i.e. the number of key values is one less than the number of children. Some of the children can be NULL(empty subtrees).

(iii) The keys in a node are in ascending order.

(iv) Keys in non leaf node will divide the left and right subtrees where value of left subtree keys will be less and value of right subtree keys will be more than that particular key.

To understand these points let us consider a node of m-way search tree of order 8.



(i) This node has the capacity to hold 7 keys and 8 children.

(ii) It currently has 5 key values and 6 children. While programming we can take a variable *count* to keep track of the number of keys that a node currently holds.

(iii) $k_1 < k_2 < k_3 < k_4 < k_5$

(iv) The key k_1 is greater than all the keys in subtree pointed to by pointer C_0 and less than all the keys in subtree pointed to by pointer C_1. Similarly this relation holds true for other keys also.

$keys(C_0) < k_1 < keys(C_1) < k_2 < keys(C_2) < k_3 < keys(C_3) < k_4 < keys(C_4) < k_5 < keys(C_5)$

The tree in figure 6.139 is a multiway search tree of order 5.

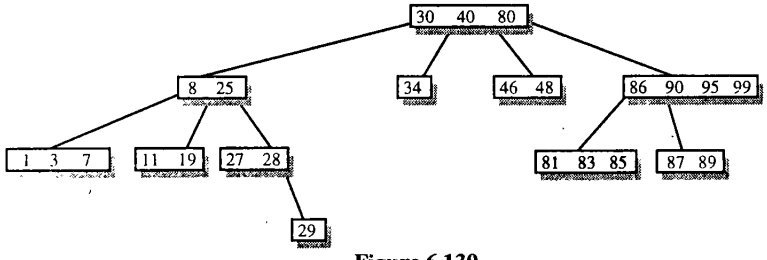


Figure 6.139

From the above explanation we can say that m-way search trees are generalized form of Binary search trees and a Binary search tree can be considered as an m-way search tree of order 2.

6.21 B-tree

In external searching, our aim is to minimize the file accesses, and this can be done by reducing the height of the tree. The height of m-way search tree is less because of its large branching factor but its height can still be reduced if it is balanced. So a new tree structure was developed (by Bayer and McCreight in 1972) which was a height Balanced m-way search tree and was named B-tree.

A B-tree of order m can be defined as an m-way search tree which is either empty or satisfies the following properties-

(i) All leaf nodes are at the same level.

(ii) All non leaf nodes (except root node) should have at least $\lceil m/2 \rceil$ children.

(iii) All nodes (except root node) should have at least $\lceil m/2 \rceil - 1$ keys.

(iv) If the root node is a leaf node(only node in the tree), then it will have no children and will have at least one key. If the root node is a non leaf node, then it will have at least 2 children and at least one key.

(v) A non leaf node with n-1 key values should have n non NULL children.

From the definition we can see that any node(except root) in a B-tree is at least half full and this avoids wastage of storage space. The B-tree is perfectly balanced so the number of nodes accessed to find a key becomes less.

The following figure shows the minimum and maximum number of children in any non root and non leaf node of B-trees of different orders.

Order of the tree	Minimum Children	Maximum Children
3	2	3
4	2	4
5	3	5
6	3	6
7	4	7
.....	.....	.....
M	⌈M/2⌉	M

Now let us see why the m-way search tree in the previous figure 6.139 is not a B-tree.

(i) The leaf nodes in this tree are [1,3,7], [11,19], [29], [34], [46,48], [81,83,85], [87,89] and it can be clearly seen that they are not at the same level.

(ii)The non leaf node [27, 28] has 2 keys but only one non NULL child, and the non leaf node [86,90,95,99] has 4 keys but only 2 non NULL children.

(iii) The minimum number of keys for a B-tree of order 5 is ⌈5/2⌉-1 = 2 while in the above tree there are 2 nodes [34], [29] which have less than 2 keys.

The tree given in figure 6.140 is a B-tree of order 5.

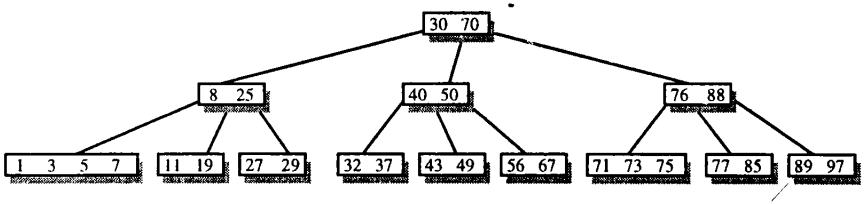


Figure 6.140

While explaining various operations on B-tree we'll take B-trees of order 5. We'll denote maximum number of permissible keys by MAX and minimum number of permissible keys(except in root) by MIN. So if the order is 5, MAX = 5-1 = 4, and MIN = ⌈5/2⌉-1 = 2.

There are special names given to B-trees of order 3 and 4. A B-tree of order 3 is known as 2-3 tree because any non root non leaf node can have 2 or 3 children, and a B-tree of order 4 is known as 2-3-4 tree because any non root non leaf node can have 2, 3 or 4 children.

6.21.1 Searching in B-tree

Searching in B tree is analogous to searching in a binary search tree. In BST, the search starts from the root and at each node we make a 2-way decision i.e. we go either to the left child or to the right child. In B tree also, search starts from the root, but here we have to make n-way decision at each node where n is the number of children of the node.

Suppose we want to search for the key 19 in the B-tree of figure 6.140. Searching will start from the root node so first we look at the node [30,70] , the key is not there and since 19 < 30, we will move on to the leftmost child of root node which is [8, 25]. The key is not present in this node also and value 19 lies between 8 and 25 so we move on to node [11,19] where we get the desired key.

If we reach a leaf node and still don't find the value, it implies that the value is not present in the tree. For example suppose we have to search for the key 35 in the tree of figure 6.140. First the key is searched in the root node, and since it lies between 30 and 70, we move to the node [40, 50]. Now 35 is less than 40, so we move to leftmost child of node [40, 50] which is [32, 37]. The key is not present in this node also, and since we have reached a leaf node the search is unsuccessful.

6.21.2 Insertion in B-tree

Firstly the key to be inserted is searched in the tree and if it is already present, we don't proceed further because duplicate keys are not allowed. If the search is unsuccessful, we will traverse down the tree and reach the leaf node where the key will be inserted. Now we can have two cases for inserting the key in the leaf node-

1. Node is not full i.e. it has less than MAX keys.
2. Node is already full i.e. it has MAX keys.

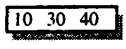
In the first case, we can simply insert the key in the node at its proper place, by shifting keys greater than it to the right side.

In the second case, the key is to be inserted into a full node; the full node is split into three parts. The first part consists of all the keys left to the median key and these keys remain in the same node. The second part consists of the median key and it is moved to the parent node. The third part consists of all the keys to the right of median key, and these keys are moved to a new node. The median key is to be moved to the parent node but what if the parent node is also full. In that case again a split will occur and this process will continue until we reach a non full parent node. In extreme case we may reach the root node which is full, so here the root node will be split and a new root will be created which will have the median key, and the height of the tree will increase by one. Let us take some keys and create a B-tree of order 5.

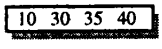
10, 40, 30, 35, 20, 15, 50, 28, 25, 5, 60, 19, 12, 38, 27, 90, 45, 48

(a) Insert 10

(b) Insert 40

(c) Insert 30 

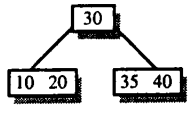
30 will be inserted between 10 and 40 since all the keys in a node should be in ascending order.

(d) Insert 35 

The maximum number of permissible keys for a node of a B-tree of order 5 is 4, so now after the insertion of 35, this node has become full.

(e) Insert 20

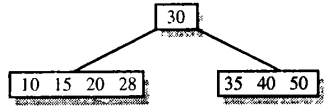
The node [10, 30, 35, 40] will become overfull [10, 20, **30**, 35, 40], so splitting is done at the median key 30. A new node is allocated and keys to the right of median key i.e. 35 and 40 are moved to the new node and the keys to the left of the median key i.e. 10 and 20 remain in the same node. Generally after splitting, the median key goes to the parent node, but here the node that is being split is the root node, so a new node is allocated which contains the median key and now this new node becomes the root of the tree.



Since the root node has been split, the height of the tree has increased by one.

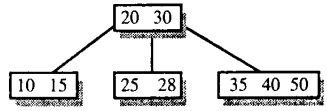
(f) Insert 15, 50, 28

15 and 28 are less than 30 so they are inserted in the left node at appropriate place and 50 is greater than 30 so it is inserted in the right node.

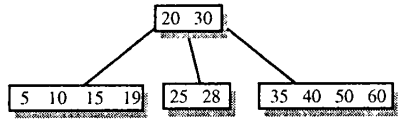


(g) Insert 25

The node [10, 15, 20, 28] will become overfull [10, 15, **20**, 25, 28], so splitting is done and the median key 20 goes to the parent node.

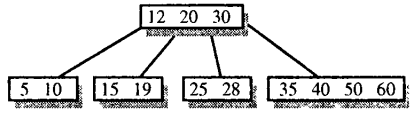


(h) Insert 5, 60, 19



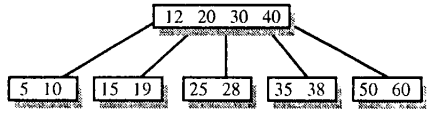
(i) Insert 12

The node [5, 10, 15, 19] will become overfull [5, 10, **12**, 15, 19], so splitting is done and the median key 12 goes to the parent node.

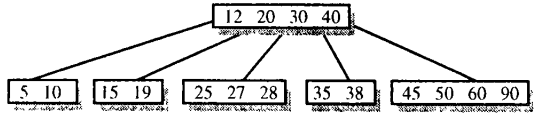


(j) Insert 38

The node[35, 40, 50, 60] will become overfull [35, 38, **40**, 50, 60], so splitting is done and the median key 40 goes to the parent node.

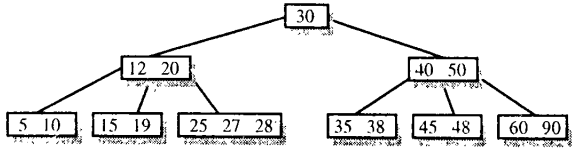


(k) Insert 27, 90, 45



(l) Insert 48

The node [45, 50, 60, 90] will become overfull [45, 48, **50**, 60, 90], so splitting is done and the median key 50 goes to the parent node. After insertion of 50 the parent node also becomes overfull [12, 20, **30**, 40, 50] so again splitting is done and this time root node is splitted so a new root is formed and the tree becomes taller.



6.21.3 Deletion in B-tree

While inserting a key, we had to take care that the number of keys should not exceed MAX, similarly while deleting we have to watch that the number of keys in a node should not become less than MIN. While inserting, when the keys exceeded MAX we splitted the node into two nodes and median key went to the parent node; while deleting when the keys will become less than MIN we will combine two nodes into one.

Now let us see how all this is done by studying all the cases of deletion. Deletion in a B-tree can be classified into two cases-

(A) Deletion from leaf node.

(B) Deletion from non leaf node.

The figure 6.141 shows all the cases of deletion.

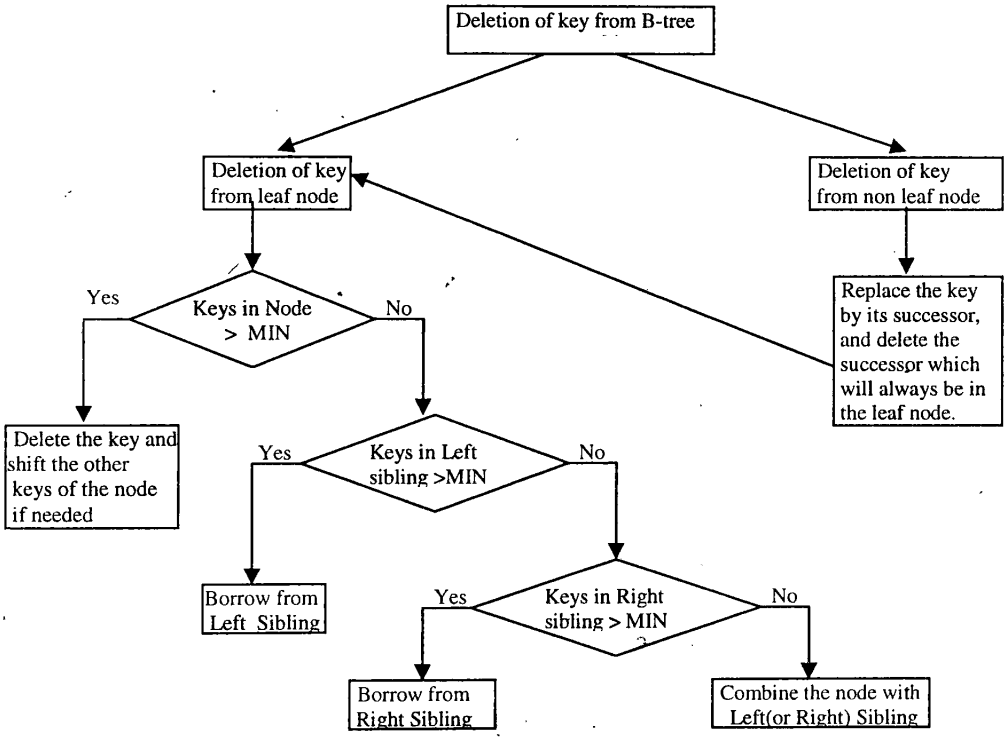

Deletion of key from B-tree
Deletion of key from leaf node
Deletion of key from non leaf node
Yes
Keys in Node > MIN
No
Replace the key by its successor, and delete the successor which will always be in the leaf node.
Delete the key and shift the other keys of the node if needed
Yes
Keys in Left sibling >MIN
No
Borrow from Left Sibling
Yes
Keys in Right sibling > MIN
No
Borrow from Right Sibling
Combine the node with Left(or Right) Sibling

Figure 6.141

6.21.3.1 Deletion from leaf node

6.21.3.1.1 If node has more than MIN keys.

In this case, deletion is very simple and key can be very easily deleted from the node by shifting other keys of the node.

(a) Delete 7, 52 from tree in figure 6.142

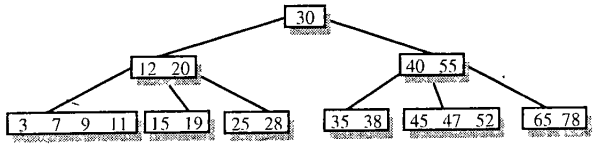
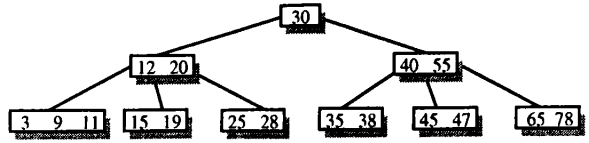


Figure 6.142



9 and 11 can be shifted left to fill the gap created by the deletion of 7. The key 52 is the rightmost key so other keys in the node need not be shifted.

6.21.3.1.2 If node has MIN keys

After deletion of a key the node will have less than MIN keys, and will become underflow node. In this case we can borrow a key from the left or right sibling if anyone of them has more than MIN keys. In our algorithm we'll first try to borrow a key from the left sibling, if the left sibling has only MIN keys then we'll try to borrow from the right sibling. If both siblings have MIN keys, then we will apply another procedure discussed later.

When a key is borrowed from the left sibling, the separator key in the parent is moved to the underflow node and the last key from the left sibling is moved to the parent.

When a key is borrowed from the right sibling, the separator key in the parent is moved to the underflow node and the first key from the right sibling is moved to the parent. All the remaining keys in the right sibling are moved one position left.

(b) Delete 15 from tree in figure 6.143.

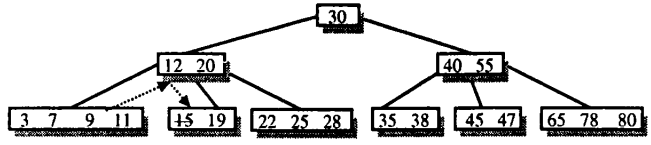


Figure 6.143

Here key 15 is to be deleted from node [15, 19], since this node has only MIN keys we will try to borrow from its left sibling [3, 7, 9, 11] which has more than MIN keys. The parent of these nodes is node [12, 20] and the separator key is 12. So the last key of left sibling(11) is moved to the place of separator key and the separator key is moved to the underflow node. The resulting tree after deletion of 15 will be-

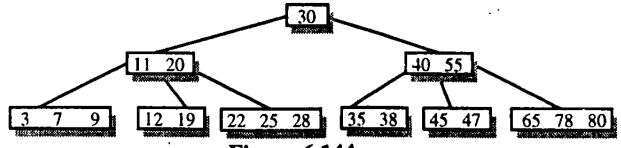
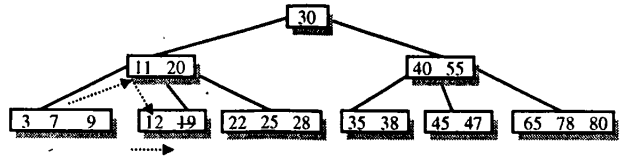


Figure 6.144

The involvement of separator key is necessary because if we simply borrow 11 and put it at the place of 15, then the basic definition of B-tree will be violated.

(c) Delete 19 from the tree in figure 6.144



We will borrow from the left sibling so key 9 will be moved up to the parent node and 11 will be shifted to the underflow node. In the underflow node the key 12 will be shifted to the right to make place for 11. The resulting tree is-

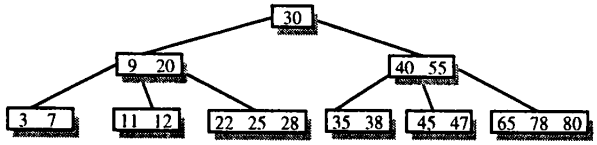
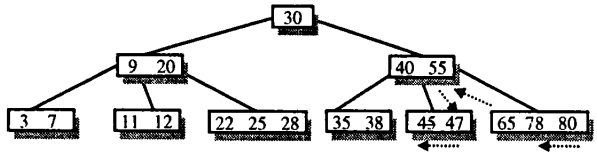
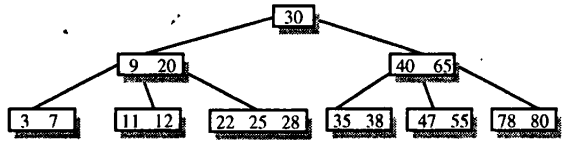


Figure 6.145

(d) Delete 45 from tree in figure 6.145



The left sibling of [45,47] is [35,38] which has only MIN keys so we can't borrow from it, hence we will try to borrow from the right sibling [65, 78, 80]. The first key of the right sibling(65) is moved to the parent node and the separator key from the parent node(55) is moved to the underflow node. In the underflow node, 47 is shifted left to make room for 55. In the right sibling, 78 and 80 are moved left to fill the gap created by removal of 65. The resulting tree is-



If both left and right siblings of underflow node have MIN keys, then we can't borrow from any of the siblings. In this case, the underflow node is combined with its left (or right) sibling.

(e) Delete 28 from the tree in figure 6.146.

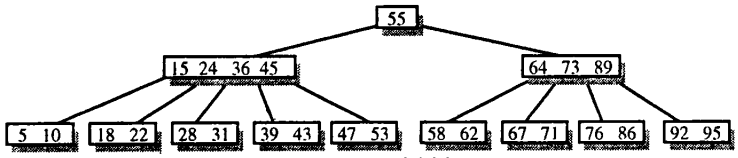
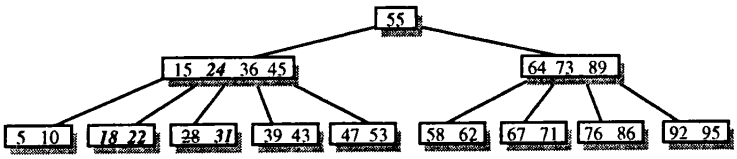


Figure 6.146

We can see that the node [28,31] has only MIN keys so we'll try to borrow from left sibling [18, 22], but it also has MIN keys so we'll look at the right sibling [39,43] which also has only MIN keys. So after deletion of 28 we'll combine the underflow node with its left sibling. For combining these two nodes the separator key(24) from the parent node will move down in the combined node.



The resulting tree after deletion of 28 is-

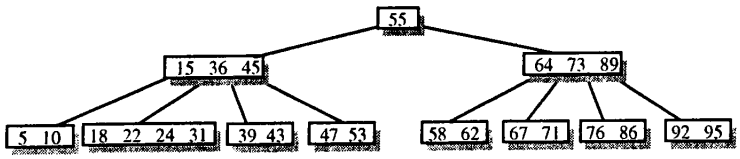
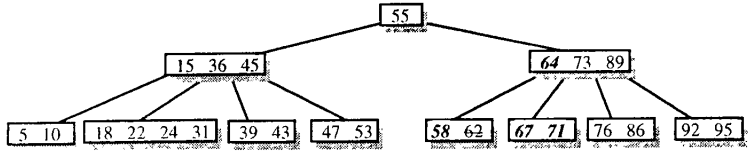


Figure 6.147

(f) Delete 62 from the tree in figure 6.147.



Here the key is to be deleted from [58, 62] which is leftmost child of its parent, and hence it has no left sibling. So here we'll look at the right sibling for borrowing a key, but the right sibling has only MIN keys, so we'll delete 62 and combine the underflow node with the right sibling. The resulting tree after deletion of 62 is-

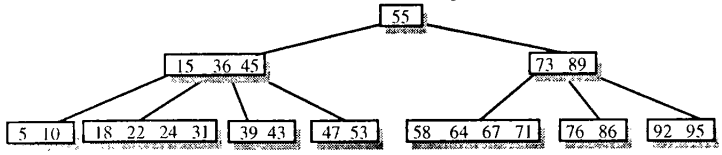
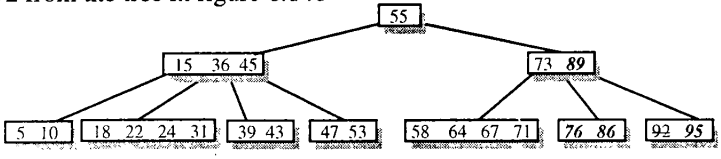
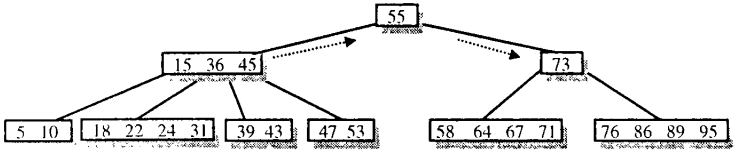


Figure 6.148

(g) Delete 92 from the tree in figure 6.148



After deletion of 92, the underflow node is combined with its left sibling.



After combining the two nodes, the parent node becomes underflow[73], so we will borrow a key from its left sibling [15,36,45]. After borrowing the resulting tree is-

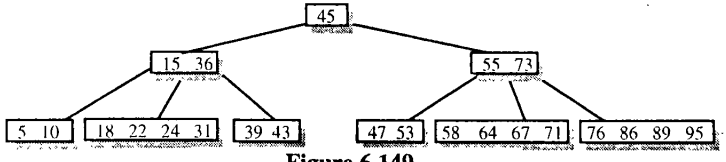


Figure 6.149

Note that before borrowing, the rightmost child of left sibling was [47,53] and after borrowing this node becomes leftmost child of node [55,73].

(h) Delete 22, 24 from the tree in figure 6.149.

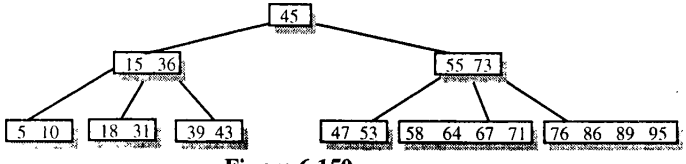
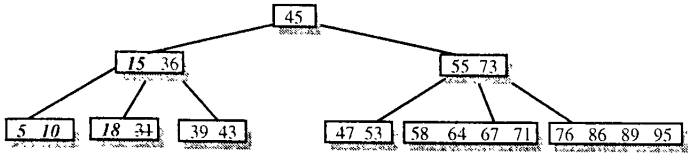
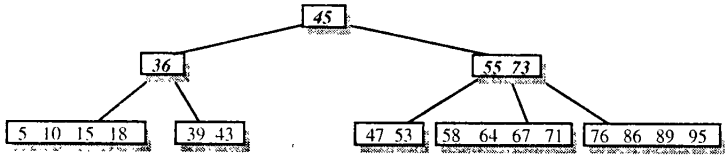


Figure 6.150

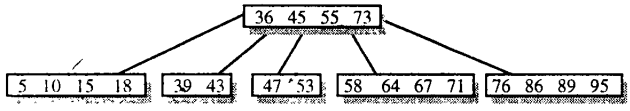
(i) Delete 31 from the tree in figure 6.150.



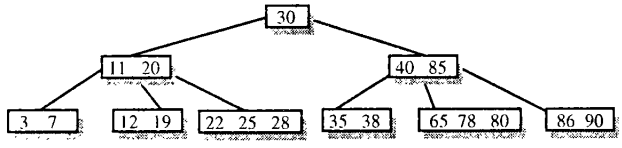
The underflow node [18] is combined with its left sibling.



Now the parent node [36] has become underflow so we will try to borrow a key from its right sibling(since it is leftmost node and has no left sibling), but the right sibling has MIN keys so we will combine the underflow node[36] with its right sibling[55, 73]. The separator key(45) comes down in the combined node, and since it was the only key in the root node, now the root node becomes empty and the combined node becomes the new root of the tree and height of the tree decreases by one. The resulting tree is-



These were some examples of deletion from a leaf node. Note that if the key is to be deleted from a node which is the leftmost child of its parent, then no left sibling exists for it so we'll consider only the right sibling for borrowing or combining. For example consider this tree-



If we have to delete a key from [3,7] then we'll have to combine it with its right sibling [12,19], and if we have to delete a key from [35, 38], we have to borrow from its right sibling [65, 78, 80].

6.21.3.2 Deletion from non leaf node.

In this case, the successor key is copied at the place of the key to be deleted and then the successor is deleted. Successor key is the smallest key in the right subtree and will always be in the leaf node. So this case reduces to case A of deletion from a leaf node.

(j) Delete 12 from the tree in figure 6.151.

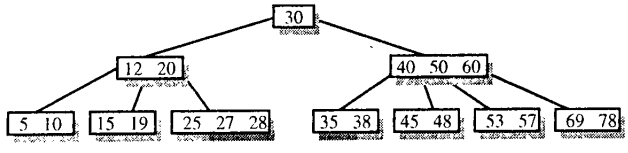
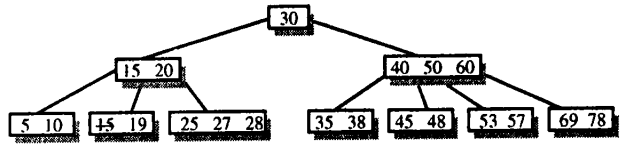


Figure 6.151

The successor key of 12 is 15, so we'll copy 15 at the place of 12 and now our task reduces to deletion of 15 from the leaf node. This deletion is performed by borrowing a key from the right sibling.



The tree after deletion of 12 is-

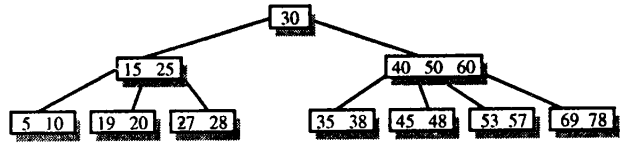
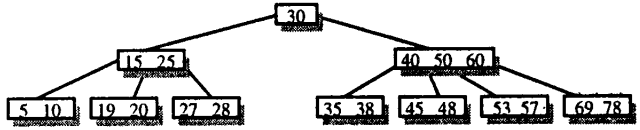
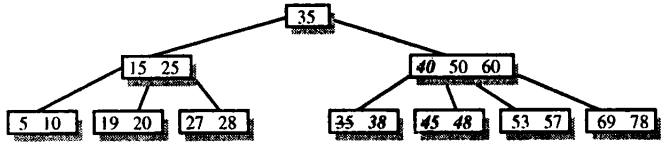


Figure 6.152

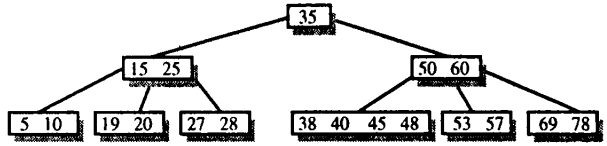
(k) Delete 30 from the tree in figure 6.152.



Successor of 30 is 35, so it is copied at the place of 30 and now 35 will be deleted from the leaf node.



The resulting tree after deletion of 30 is-



These were some examples of deletion from non leaf node. We could have taken the predecessor key also instead of successor, as predecessor key is the largest key in the left subtree and is always in the leaf node.
The `main()` function and declarations for the B tree program are-

```
/*P6.9 Program for performing various operations in a B-tree*/
#include<stdio.h>
#include<stdlib.h>
#define M 5     /*order of B tree*/
#define MAX (M-1)       /*Maximum number of permissible keys in a node*/
#if M%2==0
#define CEIL_Mdiv2 (M/2)
#else
#define CEIL_Mdiv2 ((M+1)/2)
#endif
#define MIN (CEIL_Mdiv2-1)     /*Minimum number of permissible keys in a node except root*/
struct node
{
        int count;
        int key[MAX+1];
        struct node *child[MAX+1];
};
```

```
struct node *Search(int skey,struct node *p,int *pn);
int search_node(int skey,struct node *p,int *pn);
void display(struct node *ptr,int blanks);
void inorder(struct node *ptr);
/*Functions used in insertion*/
struct node *Insert(int ikey,struct node *proot);
int rec_ins(int ikey,struct node *p,int *pk,struct node **pkrchild);
void insertByShift(int k,struct node *krchild,struct node *p,int n);
void split(int k,struct node *krchild,struct node *p,int n,int *upkey,struct node **newnode);
/*Functions used in Deletion*/
struct node *Delete(int dkey,struct node *root);
void rec_del(int dkey,struct node *p);
void delByShift(struct node *p,int n);
int copy_succkey(struct node *p,int n);
void restore(struct node *p,int n);
void borrowLeft(struct node *p,int n);
void borrowRight(struct node *p,int n);
void combine(struct node *p,int m);
int main()
{
        struct node *root = NULL, *ptr;
        int key,choice,n;
        while(1)
        {
                printf("1.Search\n2.Insert\n3.Delete\n");
                printf("4.Display\n5.Inorder traversal\n6.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:
                                printf("Enter the key to be searched : ");
                                scanf("%d",&key);
                                if( (ptr=Search(key, root, &n)) == NULL )
                                        printf("Key not present\n");
                                else
                                        printf("Node %p, Position %d\n",ptr,n);
                                break;
                        case 2:
                                printf("Enter the key to be inserted : ");
                                scanf("%d",&key);
                                root = Insert(key, root);
                                break;
                        case 3:
                                printf("Enter the key to be deleted : ");
                                scanf("%d",&key);
                                root = Delete(key, root);
                                break;
                        case 4:
                                printf("Btree is :\n\n");
                                display( root, 0 );
                                printf("\n\n");
                                break;
                        case 5:
                                inorder(root);
                                printf("\n\n");
                                break;
                        case 6:
                                exit(1);
                        default:
                                printf("Wrong choice\n");
                                break;
                }/*End of switch*/
        }/*End of while*/
```

```
}/*End of main()*/
```

The structure of a B-tree node is taken as-

```
struct node
{
        int count;
        int key[MAX+1];
        struct node *child[MAX+1];
};
```

Here `count` represents the number of keys currently present in a given node. It is incremented when a key is inserted into the node and decremented when a key is deleted from the node. We will take two arrays, one of type `int` for the keys, and the other of type `struct node*` for the children. The size of both arrays is MAX+1.

The maximum number of permissible children of a node are MAX+1 so all the elements of array `child` will be used. The maximum number of permissible keys in a node is MAX so we'll never use `key[0]`. Taking the size of array key as MAX+1 simplifies the code. In the program whenever we will have to shift keys and pointers of a node we can simply do it by using a `for` loop. Only the case of `child[0]` has to be handled separately.

The symbolic constant M represents the order of the B-tree. The maximum number of keys in a node is represented by MAX and is equal to (M-1). The minimum number of keys in a node(except root) is given by MIN which is equal to $\lceil M/2 \rceil - 1$.

6.21.4 Searching

The functions used in searching are `Search()` and `search_node()`. `Search()` is a recursive function that is used to search a key by moving down the tree using child pointers. It uses a function `search_node()` to search for the key inside the current node.

```
struct node *Search(int skey,struct node *p,int *pn)
{
        int found;
        if(p == NULL)  /*Base Case 1 : if key not found*/
                return NULL;
        found = search_node(skey,p,pn);
        if(found)        /*Base Case 2 : if key found in node p*/
                return p;
        else          /*Recursive case : Search in node p->child[*pn]*/
                return Search(skey,p->child[*pn],pn);
}/*End of Search()*/
int search_node(int skey,struct node *p,int *pn)
{
        if(skey < p->key[1]) /*skey less than leftmost key*/
        {
                *pn = 0;
                return 0;
        }
        *pn = p->count;
        while((skey < p->key[*pn]) && *pn>1)
                (*pn)--;
        if(skey == p->key[*pn])
                return 1;
        else
                return 0;
}/*End of search_node()*/
```

Let us first see how the function `search_node()` works. This function searches for `skey` in the node `p` and returns 1 if the key is present in the node otherwise it returns 0. If the key is found then `*pn` represents the position of the key in the node, otherwise the value `*pn` is used by `Search()` function to move to the appropriate child.

If `skey` is less than the leftmost key of the node then 0 is returned indicating that key is not present in this node and value of `*pn` is set to 0 which instructs the function `Search()` to continue its search in the 0^{th} child of node `*p`. If `skey` is greater than the leftmost key then we start searching for the key in the node from the right side.

`Search()` is a tail recursive function. It returns NULL if the key is not found in the tree, otherwise it returns the address of the node in which the key is found. The first argument `skey` is the key that is to be searched in the tree, pointer `p` represents the root of the tree on which the search proceeds, and the last argument is a pointer to int that will be used to give the position of key inside the node.

Recursion can stop in 2 cases, first when we reach the leaf node i.e. `p==NULL` indicating that skey is not present in the tree, and second is if `search_node()` returns 1 indicating that `skey` is present in the current node `p`.

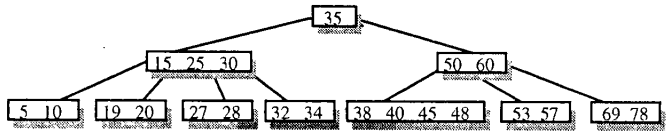
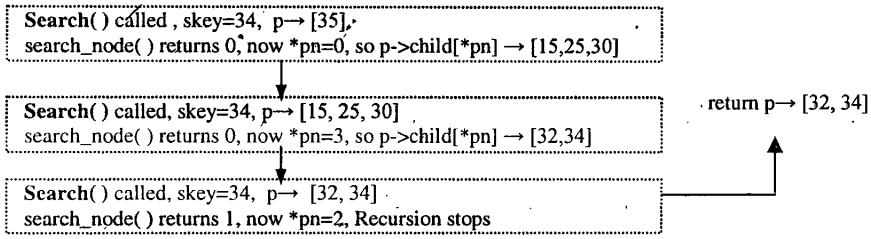


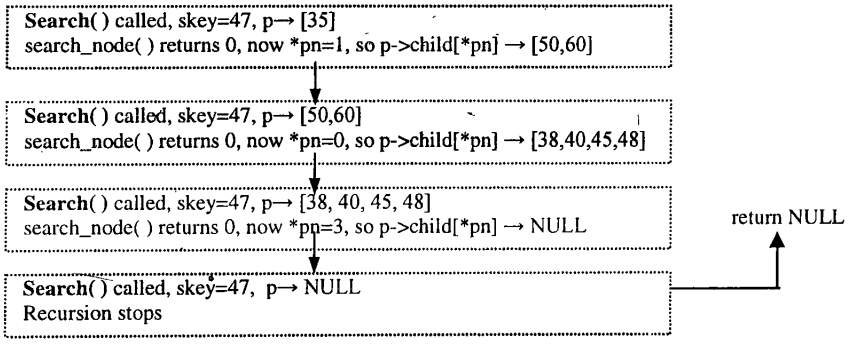
Figure 6.153

Search 34 in tree of figure 6.153


Search( ) called , skey=34, p→ [35],
search_node( ) returns 0, now *pn=0, so p->child[*pn] → [15,25,30]
Search( ) called, skey=34, p→ [15, 25, 30]
search_node( ) returns 0, now *pn=3, so p->child[*pn] → [32,34]
Search( ) called, skey=34, p→ [32, 34]
search_node( ) returns 1, now *pn=2, Recursion stops
return p→ [32, 34]

*pn=2

Search 47 in tree of figure 6.153


Search( ) called, skey=47, p→ [35]
search_node( ) returns 0, now *pn=1, so p->child[*pn] → [50,60]
Search( ) called, skey=47, p→ [50,60]
search_node( ) returns 0, now *pn=0, so p->child[*pn] → [38,40,45,48]
Search( ) called, skey=47, p→ [38, 40, 45, 48]
search_node( ) returns 0, now *pn=3, so p->child[*pn] → NULL
Search( ) called, skey=47, p→ NULL
Recursion stops
return NULL

The key 47 is not present in the tree.

The recursion used in `Search()` is tail recursion so nothing is done in the unwinding phase.

6.21.5 Insertion

The functions used in insertion are `Insert()`, `rec_ins()`, `insertByShift()` and `split()`.

```
struct node *Insert(int ikey,struct node *proot)
{
        int k,taller;
        struct node *krchild,*temp;
```

```
        taller = rec_ins(ikey,proot,&k,&krchild);
        if(taller)  /*tree grown in height, new root is created*/
        {
                temp = (struct node *)malloc(sizeof(struct node));
                temp->count = 1;
                temp->child[0] = proot;
                temp->key[1] = k;
                temp->child[1] = krchild;
                proot = temp;
        }
        return proot;
}/*End of Insert()*/
```

Here `ikey` is the key to be inserted and `proot` is pointer to the root of the tree. The function returns a pointer to the root of the tree.

Inside `Insert()`, a recursive function `rec_ins()` is called which performs the main task of inserting the key into the tree. It takes 4 arguments, the first two are the same as in `Insert()`. The last two arguments are addresses of variables `k` and `krchild` i.e. this function will set values of variables `k` and `krchild`. The return value of `rec_ins()` function will determine whether the tree has grown in height or not. A B-tree grows in height only when the root node is split (see steps (e) and (l) insertion of 20 and 48). When root node is split then a new root has to be created which needs allocation of a new node. A new node is allocated which contains the key `k` and the original root node is made its left child while `krchild` is made its right child and finally this new node is made the new root of the tree.

For example in step (l) insertion of 48, the root node before insertion was [12, 20, 30, 40], the function `rec_ins()` returns 1 and sets the value of `k` as 30 and makes `krchild` point to [40,50]. The new root node contains only one key which is `k` and its right child pointer is `krchild`, and the old root node is its left child pointer. Now let us discuss the working of this function `rec_ins()`.

The function `rec_ins()` is a recursive function. In the winding phase we move down the tree recursively. There can be two base cases, one when we reach a NULL subtree which means we have to insert the key, and the other is when we find the key in some node and the key will not be inserted. In the unwinding phase, we insert the key and perform the splitting if required. In unwinding phase we move up the tree on the path of insertion, and we know that splitting is propagated upwards so the insertion of key and splitting is done in this phase.

```
int rec_ins(int ikey,struct node *p,int *pk,struct node **pkrchild)
{
        int n;
        int flag;
        if(p==NULL)  /*Base case 1*/
        {
                *pk = ikey;
                *pkrchild = NULL;
                return 1;
        }
        if(search_node(ikey,p,&n)) /*Base Case 2*/
        {
                printf("Duplicate keys are not allowed\n");
                return 0;
        }
        flag = rec_ins(ikey,p->child[n],pk,pkrchild);
        if(flag)
        {
                if(p->count < MAX)
                {
                        insertByShift(*pk,*pkrchild,p,n);
                        return 0;
                }
                else
                {
                        split(*pk,*pkrchild,p,n,pk,pkrchild);
```

```
            return 1; /*median key to be inserted*/
        }
    return 0;
}/*End of rec_ins()*/
```

In the unwinding phase, we will insert the key whenever we get a non full node using `insertByShift()`, otherwise we will call `split()`.

The return value of `rec_ins()` can be 1 or 0. The return value of 1 indicates that the insertion is not complete and we need to continue work in the unwinding phase. The return value of 0 means that insertion is finished and there is no need to do any work in the unwinding phase.

Initially when we reach the base case(`p==NULL`), `rec_ins()` returns 1. When the function `split()` is called, 1 is returned indicating that the insertion is not over and the median key is still there waiting to be inserted in the tree. When `insertByShift()` is called, 0 is returned indicating insertion is finished.

When we have a duplicate key, there is nothing to be done in the unwinding phase because the key is not inserted in the tree. So in this case also we return 0.

`*pk` and `*pkrchild` represent the key to be inserted and its right child respectively. Initially when the recursion stops `*pk` is set to `ikey` and `*pkrchild` is set to NULL. Whenever `split()` is called , it sets the value of `*pk` to the median key and also changes the value of `*pkrchild`.

We will take some examples and see how the key is actually inserted. Let us insert key 17 in the tree given in figure 6.154..

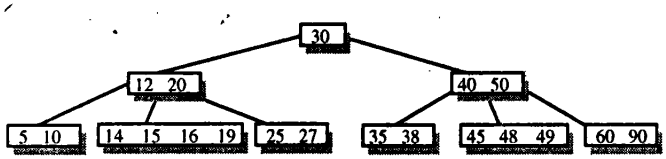
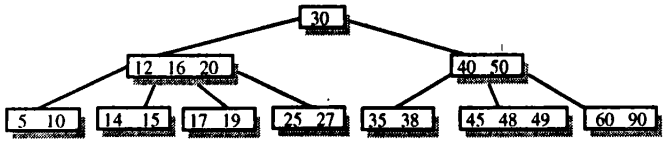


Figure 6.154

For insertion of 17, splitting is required and the resulting tree is-



Now let us trace and see how this insertion happens in our program.

1) Initially `Insert()` calls `rec_ins()` with `p` pointing to root node[30]. Inside `rec_ins()`, since `p` is not NULL, `search_node()` is called which sets n=0, and a recursive call is made to `rec_ins()` with 0^{th} child of `p` i.e. [12,20].

2) In the second recursive call of `rec_ins()`, `p` points to node [12,20] which is not NULL so `search_node()` is called which sets n=1. A recursive call is made to `rec_ins()` with 1^{st} child of `p` i.e. [14,15,16,19].

3) In the third recursive call of `rec_ins()`, `p` points to node [14,15,16,19] , `search_node()` sets n=3. Now a recursive call is made with 3^{rd} child of `p` which is NULL.

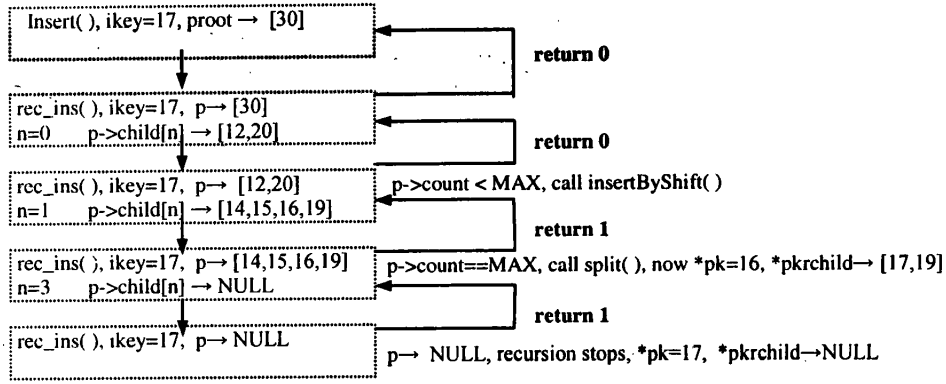
4) In the fourth recursive call of `rec_ins()` `p` is NULL, hence we have reached the base case and the recursion stops. `*pk` is set to 17, and `*pkrchild` is set to NULL which means that the key to be inserted is 17 and child to its right will be NULL. This recursive call of `rec_ins()` returns 1 and unwinding phase begins.

5) Now we go back to the 3^{rd} recursive call of `rec_ins()` where `p` points to node [14,15,16,19]. The fourth recursive call had returned 1, so value of flag is 1 which means that insertion is not yet over, so we check the number of keys in the node. Since keys are equal to MAX we need to call the function `split()`. This function splits the node and now sets `*pk` to 16, `*pkrchild` points to [17,19]. This means that now the key to be inserted is 16, and its right child will be [17,19]. The 3^{rd} recursive call finishes and it returns 1.

6) Now we go back to the 2nd recursive call of `rec_ins()` where p points to node [12, 20]. The third recursive call had returned 1, so value of flag is 1 which means that insertion is not yet over, so we check the number of keys in the node. Since keys are less than MAX, we call the function `insertByShift()`. This function inserts the key at proper place in this node. The 2nd recursive call finishes and it returns 0.

7) Now we go back to the 1st recursive call of `rec_ins()` where p points to node [30]. The second recursive call had returned 0, so value of flag is 0 which means that insertion is over. The value 0 is returned.

8) Now we go back to `Insert()`. The outermost call of `rec_ins()` returned 0 indicating that insertion is over.



Now let us insert key 95 in the tree given in figure 6.155.

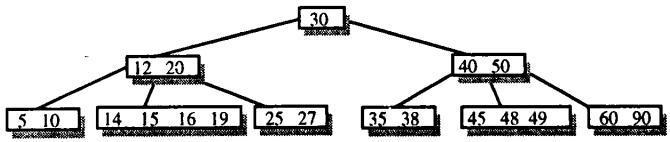
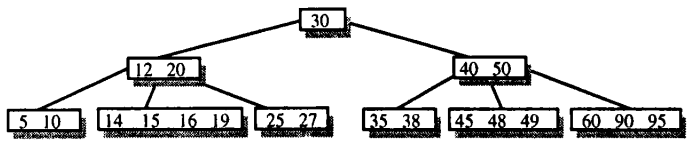
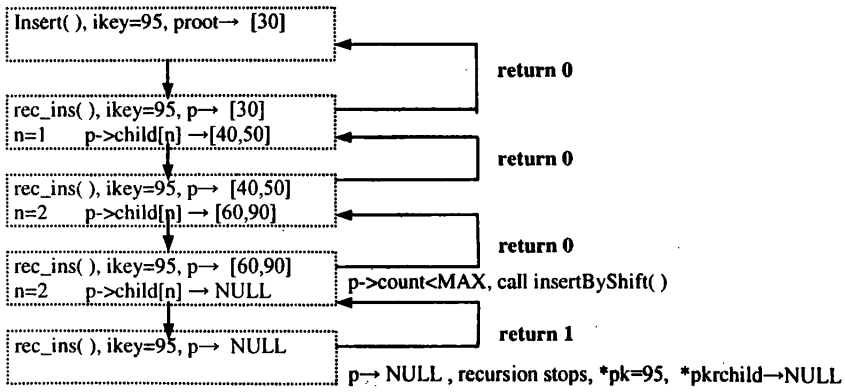


Figure 6.155

95 can be inserted in the leaf node, so no splitting is required. The resulting tree after insertion is-





Now let us insert key 80 in the tree given in figure 6.156.

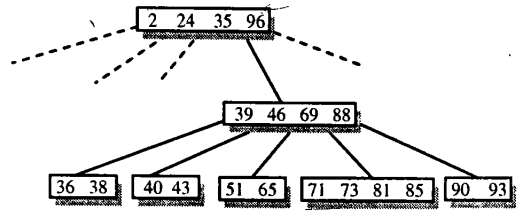
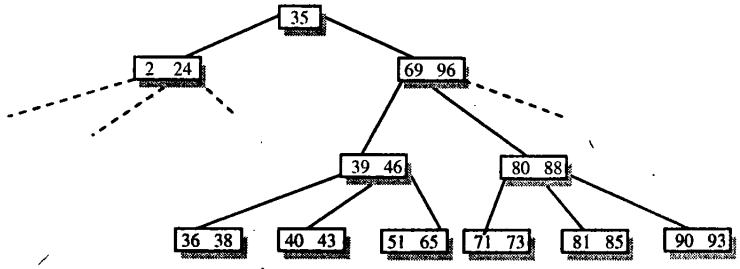
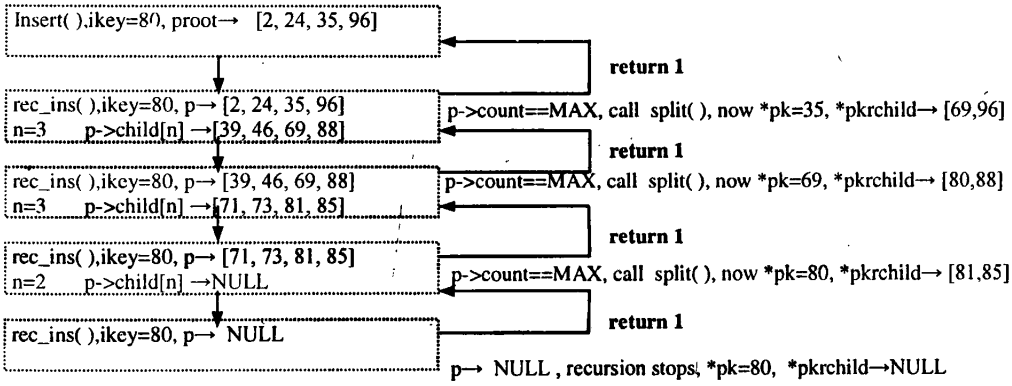


Figure 6.156

In this case, the splitting is propagated up to the root node. The root node is split and a new root is created and the height of tree increases.



Here the outermost(first) recursive call returns 1 to `Insert()` indicating that insertion has not completed and still the key 35 remains to be inserted. Inside `Insert()`, value of taller becomes 1 and a new root is created and 35 is inserted into it.


Insert( ),ikey=80, proot→ [2, 24, 35, 96]
return 1
rec_ins( ),ikey=80, p→ [2, 24, 35, 96]
n=3 p->child[n] →[39, 46, 69, 88]
p->count==MAX, call split( ), now *pk=35, *pkrchild→ [69,96]
return 1
rec_ins( ),ikey=80, p→ [39, 46, 69, 88]
n=3 p->child[n] →[71, 73, 81, 85]
p->count==MAX, call split( ), now *pk=69, *pkrchild→ [80,88]
return 1
rec_ins( ),ikey=80, p→ [71, 73, 81, 85]
n=2 p->child[n] →NULL
p->count==MAX, call split( ), now *pk=80, *pkrchild→ [81,85]
return 1
rec_ins( ),ikey=80, p→ NULL
p→ NULL , recursion stops, *pk=80, *pkrchild→NULL

Now let us see how the functions `insertByShift()` and `split()` work.

```
void insertByShift(int k,struct node *krchild,struct node *p,int n)
{
        int i;
        for(i=p->count; i>n; i--)
        {
                p->key[i+1] = p->key[i];
                p->child[i+1] = p->child[i];
        }
        p->key[n+1] = k;
        p->child[n+1] = krchild;
        p->count++;
}/*End of insertByShift()*/
```

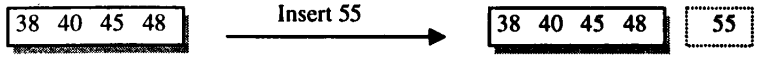
This function will be called only if node `p` contains less than MAX keys i.e. when there is no chance of overflow if a new key is inserted into it, and the key can be simply inserted by shifting some keys to the right. This function will insert the key `k` and pointer `krchild` into the node `p` at the $(n+1)^{th}$ position. For this initially

all keys and pointers which are after the n^{th} position are shifted right one position to make room for key k and krchild and after this these two are inserted at the $(n+1)^{th}$ position in the node and the count of node p is incremented.

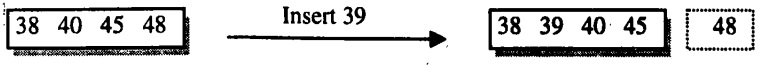
```
void split(int k,struct node *krchild,struct node *p,int n,int *upkey,
struct node **newnode)
{
        int i,j;
        int lastkey;
        struct node *lastchild;
        int d = CEIL_Mdiv2;
        if(n==MAX)
        {
                lastkey = k;
                lastchild = krchild;
        }
        else
        {
                lastkey = p->key[MAX];
                lastchild = p->child[MAX];
                for(i=p->count-1; i>n; i--)
                {
                        p->key[i+1] = p->key[i];
                        p->child[i+1] = p->child[i];
                }
                p->key[i+1] = k;
                p->child[i+1] = krchild;
        }
        *newnode = (struct node *)malloc(sizeof(struct node));
        *upkey = p->key[d];
        for(i=1,j=d+1; j<=MAX; i++,j++)
        {
                (*newnode)->key[i] = p->key[j];
                (*newnode)->child[i] = p->child[j];
        }
        (*newnode)->child[0] = p->child[d];
        p->count = d-1;   /*Number of keys in the left splitted node*/
        (*newnode)->count = M-d;  /*Number of keys in the right splitted node*/
        (*newnode)->key[M-d] = lastkey;
        (*newnode)->child[M-d] = lastchild;
}/*End of split()*/
```

This function will be called only if node p contains MAX keys.

The structure of a node cannot accommodate values more than MAX, so when a node becomes overflow we'll save the last key value into another variable named lastkey. If the new key is to be inserted at last then that new key is stored in the variable lastkey. For example here 55 is the key to be inserted and its place among all the keys is last so it is saved in variable lastkey.



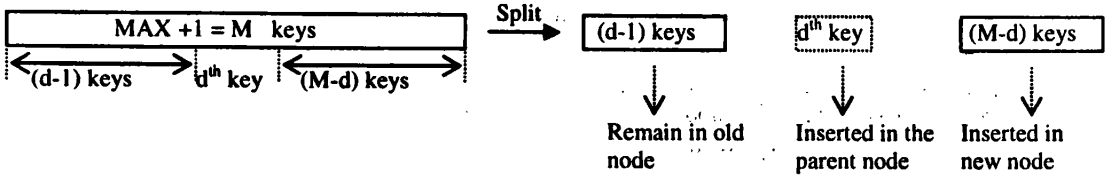
If the new key is to be inserted somewhere in between the node, then firstly the last key of the node is saved in variable lastkey and then the new key is inserted by shifting other keys to the right. For example if 39 is the key to be inserted then, first 48 is saved in variable lastkey and then 40, 45 are shifted right and 39 is inserted in the node.



The pointer value is also simultaneously saved in variable lastchild.

Now space for a new node is allocated. The variable d denotes the median value ceil(M/2). The median key i.e. the d^{th} key is the key that will be moved up to the parent so it is stored in upkey.

When a node is full it contains MAX keys, and after the arrival of a new key, the total number of keys becomes (MAX+1). This number is equal to M which is the order of the tree. Now we have to split these M keys. The first (d-1) keys which are to the left of d^{th} key remain in the same node, and the remaining M-d keys which are to the right of d^{th} key will move into a new node.



6.21.6 Deletion

The functions used in deletion are - `Delete()`, `rec_del()`, `delByshift()`, `copysucc()`, `restore()`, `borrowLeft()`, `borrowRight()`, `combine()`.

```
struct node *Delete(int dkey,struct node *root)
{
        struct node *temp;
        rec_del(dkey,root);
        /*If Tree becomes shorter, root is changed*/
        if(root!=NULL && root->count == 0)
        {
                temp = root;
                root = root->child[0];
                free(temp);
        }
        return root;
}/*End of Delete()*/
```

`Delete()` function calls another recursive function `rec_del()` which performs the main deletion process. We know that after the deletion, if the root node becomes empty then the height of the tree decreases (see step (i), deletion of 31). So, after `rec_del()` has deleted the key from the tree, `Delete()` checks whether the root node has become empty or not, and if there is no key left in the root node then the 0^{th} child of the root becomes the new root. The function `rec_del()` is-

```
void rec_del(int dkey,struct node *p)
{
        int n,flag,succkey;
        if(p==NULL)     /*reached leaf node, key does not exist*/
                printf("Value %d not found\n",dkey);
        else
        {
                flag = search_node(dkey,p,&n);
                if(flag)/*found in current node p*/
                {
                        if(p->child[n]==NULL)/*node p is a leaf node*/
                                delByShift(p,n);
                        else /*node p is a non leaf node*/
                        {
                                succkey = copy_succkey(p,n);
                                rec_del(succkey,p->child[n]);
                        }
                }
                else    /*not found in current node p*/
                        rec_del(dkey,p->child[n]);

                if(p->child[n] != NULL)/*if p is not a leaf node*/
```

```
            {
              if(p->child[n]->count < MIN)/*check underflow in p->child[n]*/
                    restore(p,n);
            }
      }
}/*End of rec_del()*/
```

In `rec_del()` there are two recursive calls, in both of them the second argument is same i.e. `p->child[n]` where n is obtained by `search_node()`. This procedure of traversing down the tree through n^{th} child of p is same as in `rec_ins()`. If `rec_del()` is called with p as NULL, then it means that the key is not present in the tree.

The function `search_node()` is called which searches for the key in the current node, if it is found then we check whether the current node is a leaf node or a non leaf node. If it is a leaf node then the key is simply deleted by shifting the other keys using function `delByShift()`, and if it is a non leaf node then the successor key is copied at the place of dkey using `copy_succ()`. This function returns successor key which is stored in variable `succkey`, and now our task is to delete this successor key, so now `rec_del()` is called with the first argument as `succkey`.

When we call `delByShift()`, underflow might occur in the node. We will check for this underflow when we return from the recursive calls i.e. in the unwinding phase. If there is underflow we'll call `restore()` which performs all the processes of borrowing keys or combining nodes. This function will need the parent of the underflow node. This is why in the code we are checking count of `p->child[n]` and not that of p. If `p->child[n]` underflows we will send its parent to the `restore()` function. If p is a leaf node then underflow is not checked because in this case `p->child[n]` will be NULL.

This process of checking underflow will check underflow in all nodes that come in the unwinding phase except root node. The root node is different from other nodes since the minimum number of keys for root node is 1. If the root node underflows i.e. it becomes empty then `Delete()` function will handle this case of underflow of root node.

Consider the tree given in figure 6.157 and delete 19 from it.

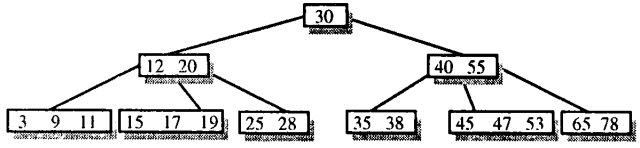
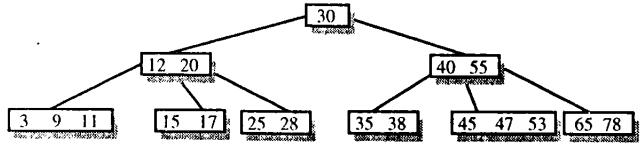
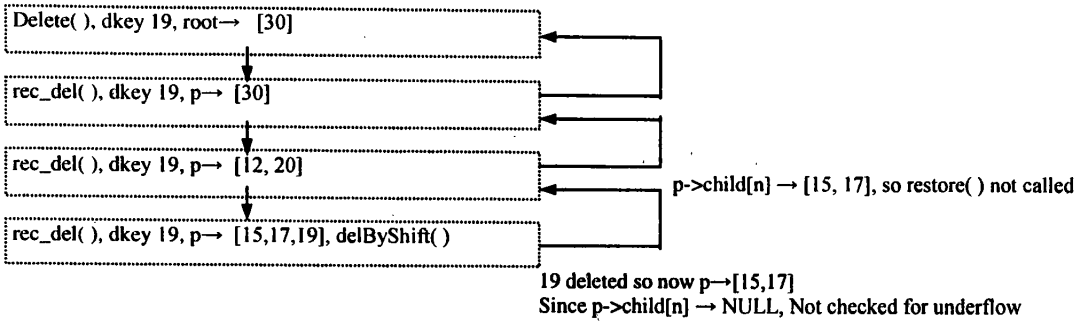


Figure 6.157

Deletion of 19 is simple since the node [15, 17, 19] contains more than MIN keys. The tree after deletion is-



When `rec_del()` is called with p→[15, 17, 19], `search_node()` returns 1 as 19 is present in this node, and so the value of flag becomes 1. Now since node p is leaf node, `delByShift()` is called and 19 is deleted from the node. This is the base case of recursion, so now recursion terminates and unwinding phase starts. In the unwinding phase, we will check whether underflow has occurred in any node or not.


Delete( ), dkey 19, root→ [30]
rec_del( ), dkey 19, p→ [30]
rec_del( ), dkey 19, p→ [12, 20]
rec_del( ), dkey 19, p→ [15,17,19], delByShift( )
p->child[n] → [15, 17], so restore( ) not called
19 deleted so now p→[15,17]
Since p->child[n] → NULL, Not checked for underflow

Consider the tree given in figure 6.158 and delete 69 from it.

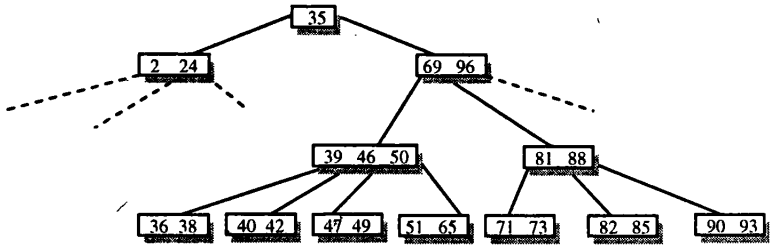
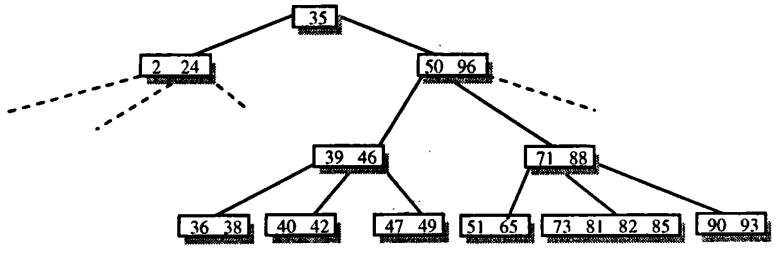
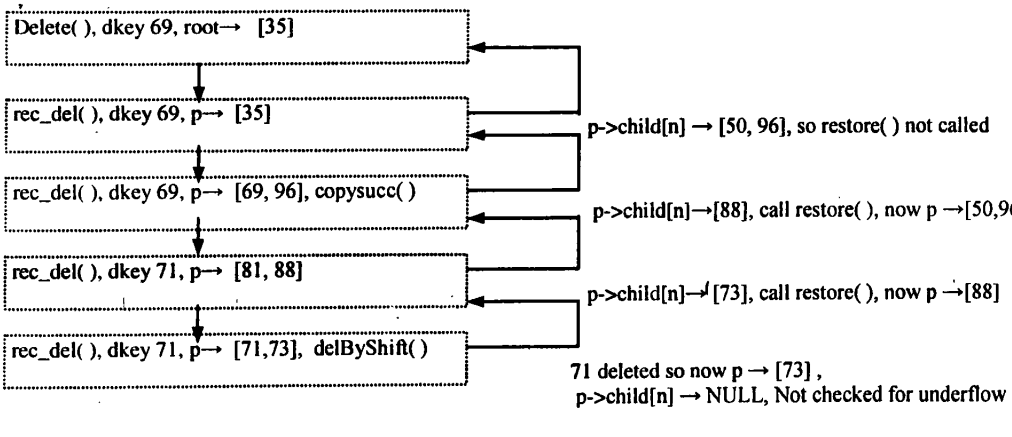


Figure 6.158

Here key 69 is present in a non leaf node, so successor key 71 will be copied at its place and then 71 will be deleted from the leaf node. The resulting tree after deletion is-




Delete( ), dkey 69, root→ [35]
rec_del( ), dkey 69, p→ [35]
rec_del( ), dkey 69, p→ [69, 96], copysucc( )
rec_del( ), dkey 71, p→ [81, 88]
rec_del( ), dkey 71, p→ [71,73], delByShift( )
p->child[n] → [50, 96], so restore( ) not called
p->child[n]→[88], call restore( ), now p →[50,96]
p->child[n]→[73], call restore( ), now p →[88]
71 deleted so now p → [73] ,
p->child[n] → NULL, Not checked for underflow

(iii) Consider the tree given next and delete 45 from it.

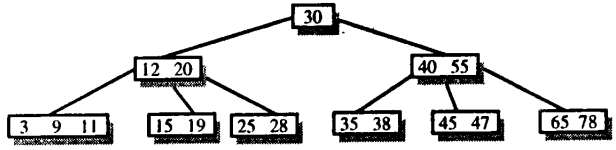
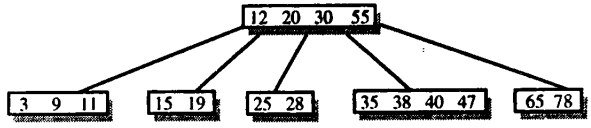
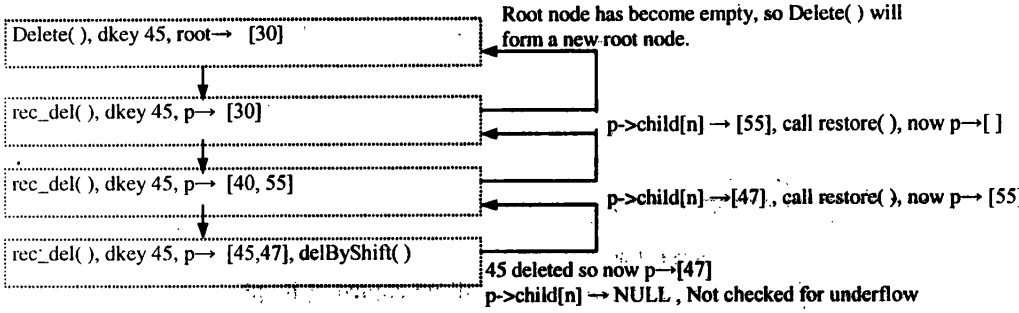


Figure 6.159

After deletion of 45, the height of the tree decreases. The resulting tree after deletion is-




Delete( ), dkey 45, root→ [30]
Root node has become empty, so Delete( ) will form a new root node.
rec_del( ), dkey 45, p→ [30]
p->child[n] → [55], call restore( ), now p→[ ]
rec_del( ), dkey 45, p→ [40, 55]
p->child[n] →[47], call restore( ), now p→ [55]
rec_del( ), dkey 45, p→ [45,47], delByShift( )
45 deleted so now p→[47]
p->child[n] → NULL , Not checked for underflow

The function `copy_succkey()` is-

```
int copy_succkey(struct node *p, int n)
{
        struct node *ptr;
        ptr = p->child[n]; /*point to the right subtree*/
        while( ptr->child[0]!=NULL )   /*move down till leaf node arrives*/
                ptr = ptr->child[0];

        p->key[n] = ptr->key[1];
        return ptr->key[1];
}/*End of copy_succkey()*/
```

This function will be required when a deletion is to be performed in a non leaf node. The task of this function is to replace the n^{th} key of node p i.e. `p->key[n]` by its successor key and return the value of that successor key. We know that the successor key is the leftmost key in the right subtree. We'll take a pointer `ptr` and use it to move down the right subtree and since we want to reach the leftmost key, we will move only leftwards using the leftmost child `child[0]`. We will stop when we reach the leaf node and the leftmost key of this node is the successor key of `p->key[n]`.

The function `delByShift()` is-

```
void delByShift(struct node *p, int n)
{
        int i;
        for(i=n+1; i <= p->count; i++)
        {
                p->key[i-1] = p->key[i];
                p->child[i-1] = p->child[i];
        }
        p->count--;
}/*End of delByShift()*/
```

This function will be called only if node p contains more than MIN keys i.e. when there is no chance of underflow if a key is deleted from the node, and the key can be simply deleted by shifting some keys to the left. This function will delete the n^{th} key and its right child pointer from the node p i.e. `key[n]` and `child[n]` are removed from this node. For this, all the keys and pointers which are to the right of n^{th} position are shifted one position left and the count of the node p is decremented.

The function `restore()` is-

```
void restore(struct node *p, int n)
{
        if(n!=0  &&  p->child[n-1]->count > MIN)
                borrowLeft(p, n);
        else if(n!=p->count && p->child[n+1]->count > MIN)
                borrowRight(p, n);
        else
        {
                if(n==0)  /*if underflow node is leftmost node*/
                        combine(p, n+1);       /*combine with right sibling*/
                else
                        combine(p, n); /*combine with left sibling*/
        }
}/*End of restore()*/
```

The function `restore()` is called when a node underflows. The underflow node is the n^{th} child of the node p. Let us recall how we proceed in the case of an underflow. First we try to borrow from left sibling and then from right sibling. If borrowing is not possible then we combine the node with left sibling, and if left sibling doesn't exist then we combine it with right sibling.

Since underflow node is the n^{th} child of the node p, $(n-1)^{th}$ child of p is the left sibling and $(n+1)^{th}$ child of p is the right sibling of underflow node. The left sibling will not exist if the underflow node is leftmost child of its parent `(n==0)`, and the right sibling will not exist if the underflow node is the rightmost child of its parent `(n==p->count)`. So before borrowing we have to check for these two conditions also. If the underflow node is not the leftmost child and the left sibling contains more than MIN keys then we can borrow from left, otherwise if the underflow node is not the rightmost child and the right sibling contains more than MIN keys then we can borrow from right. If both these conditions fail we'll combine the node with the left sibling, but if the node is leftmost we'll combine it with the right sibling.

When we combine with right sibling the second argument is n+1, and with left sibling it is n. While combining with left sibling, the second argument is not n-1, we'll come to know the reason for this after studying function `combine()`.

The function `borrowLeft()` is-

```
void borrowLeft(struct node *p, int n)
{
        int i;
        struct node *u; /* underflow node*/
        struct node *ls; /*left sibling of node u*/
        u = p->child[n];
        ls = p->child[n-1];
        /*Shift all the keys and pointers in underflow node u one position right*/
        for(i=u->count; i>0; i--)
        {
                u->key[i+1] = u->key[i];
                u->child[i+1] = u->child[i];
        }
        u->child[1] = u->child[0];
        /*Move the separator key from parent node p to underflow node u*/
        u->key[1] = p->key[n];
        u->count++;
        /*Move the rightmost key of node ls to the parent node p*/
        p->key[n] = ls->key[ls->count];
        /*Rightmost child of ls becomes leftmost child of node u */
```

```
        u->child[0] = ls->child[ls->count];
        ls->count--;
}/*End of borrowLeft()*/
```

The node u represents the underflow node, ls is its left sibling and p is their parent node. Initially all the keys and pointers in node u are shifted one position right to make room for the new key(in the figure 6.160, key 20 is shifted right). After this the separator key(18) from the parent is moved to the underflow node. The rightmost key of the left sibling(14) is moved to the parent node. The rightmost child of ls (node [15,16]) becomes the leftmost child of node u.

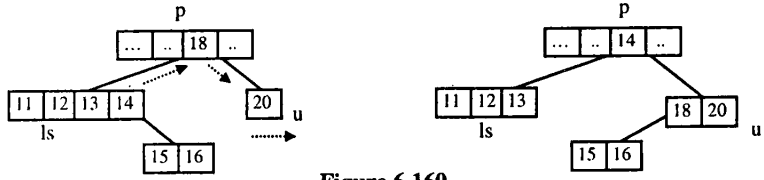


Figure 6.160

All keys and children are not shown in this figure.

The function borrowRight() is -

```
void borrowRight(struct node *p,int n)
{
        int i;
        struct node *u;         /*underflow node*/
        struct node *rs;        /*right sibling of node u*/
        u = p->child[n];
        rs = p->child[n+1];
       /*Move the separator key from the parent node p to the underflow node u*/
        u->count++;
        u->key[u->count] = p->key[n+1];
        /*Leftmost child of node rs becomes the rightmost child of node u*/
        u->child[u->count] = rs->child[0];
        /*Move the leftmost key from node rs to parent node p*/
        p->key[n+1] = rs->key[1];
        rs->count--;
        /*Shift all the keys and pointers of node rs one position left*/
        rs->child[0] = rs->child[1];
        for(i=1; i<=rs->count; i++)
        {
                rs->key[i] = rs->key[i+1];
                rs->child[i] = rs->child[i+1];
        }
}/*End of borrowRight()*/
```

The node u represents the underflow node, rs is its right sibling and p is the parent node. Initially move the separator key(in the figure 6.161, key 6) from the parent node p to the underflow node u. Note that here shifting of keys in the underflow node is not needed. The leftmost child of rs (node [7,9]) becomes the rightmost child of u. The leftmost key from node rs (10) is moved to the parent node p. At last all the remaining keys and pointers in node rs(11,12,13) are shifted one position left to fill the gap created by removal of key 10.

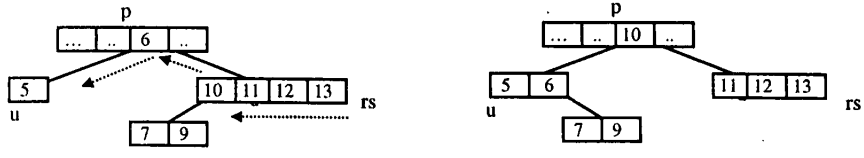


Figure 6.161

All keys and children are not shown in this figure.

The function combine() is-

```
void combine(struct node *p, int m)
{
        int j;
        struct node *x;
        struct node *y;
        x = p->child[m];
        y = p->child[m-1];
        /*Move the key from the parent node p to node y*/
        y->count++;
        y->key[y->count] = p->key[m];
        /*Shift the keys and pointers in p one position left to fill the gap*/
        for(i=m; i<p->count; i++)
        {
                p->key[i] = p->key[i+1];
                p->child[i] = p->child[i+1];
        }
        p->count--;
        /*Leftmost child of x becomes rightmost child of y*/
        y->child[y->count] = x->child[0];
        /*Insert all the keys and pointers of node x at the end of node y*/
        for(i=1; i<=x->count; i++)
        {
                y->count++;
                y->key[y->count] = x->key[i];
                y->child[y->count] = x->child[i];
        }
        free(x);
}/*End of combine()*/
```

This function combines the two nodes x and y. The node x is the m^{th} child and node y is the $(m-1)^{th}$ child of node p. Initially the separator key(key e in the figure 6.162) from the parent node is moved to node y, and all the keys that were on the right side of key e in the node p are shifted left to fill the gap created by the removal of e. Now the leftmost child of x(node [f,g]) becomes the rightmost child of y. At last all the keys and pointers of node x are inserted at the end of node y and the memory occupied by node x is released using `free()`.

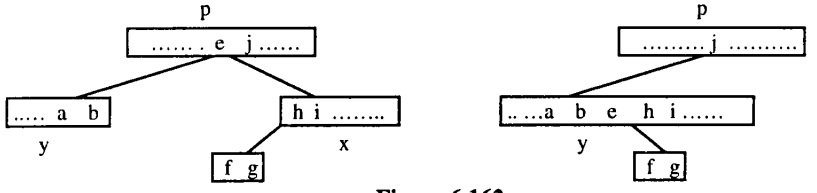


Figure 6.162

In the function `restore()` we have used combine like this-

```
{
        if(n==0)   /*if underflow node is leftmost node*/
                combine(p,n+1); /*combine n^th child of p with its right sibling*/
        else
                combine(p,n);  /*combine n^th child of p with its left sibling*/
}
```

`combine(p,n)`
m=n, x is n^{th} node, y is $(n-1)^{th}$ node
Here x is underflow node and y is its left sibling.

`combine(p,n+1)`
m=n+1, x is $(n+1)^{th}$ node and y is n^{th} node
Here y is underflow node and x is its right sibling.

The function inorder() is-

```
void inorder(struct node *ptr)
{
        int i;
        if(ptr!=NULL)
        {
                for(i=0; i<ptr->count; i++)
                {
                        inorder(ptr->child[i]);
                        printf("%d\t",ptr->key[i+1]);
                }
                inorder(ptr->child[i]);
        }
}/*End of inorder()*/
```

This function prints the inorder traversal of the B-tree. In inorder traversal, key is processed after its left subtree and before its right subtree. The left subtrees of the keys are processed in the first recursive call. The rightmost subtree of the current node is traversed by the second recursive call outside the for loop.

The function display() is-

```
void display(struct node *ptr,int blanks)
{
        if(ptr)
        {
                int i;
                for(i=1; i<=blanks; i++)
                        printf(" ");
                for(i=1; i<=ptr->count; i++)
                        printf("%d ",ptr->key[i]);
                printf("\n");
                for(i=0; i<=ptr->count; i++)
                        display(ptr->child[i],blanks+10);
        }
}/*End of display()*/
```

6.22 B+ tree

A disadvantage of B tree is inefficient sequential access. If we want to display the data in ascending order of keys, we can do an inorder traversal but it is time consuming, let us see the reason for it. While doing inorder traversal, we have to go up and down the tree several times, i.e. the nodes have to be accessed several times. Whenever an internal node is accessed, only one element from it is displayed and we have to go to another node. We know that each node of a B tree represents a disk block and so moving from one node to another means moving from one disk block to another which is time consuming. So for efficient sequential access, the number of node accesses should be as few as possible.

B+ tree which is a variation of B tree, is well suited for sequential access. The two main differences in B tree and B+ tree are-

(i) In B tree, all the nodes contain keys, their corresponding data items (records or pointers to records), and child pointers but in B+ tree the structures of leaf nodes and internal nodes are different. The internal nodes store only keys and child pointers while the leaf nodes store keys and their corresponding data items. So the data items are present only in the leaf nodes. The keys in the leaf nodes may also be present in the internal nodes.

(ii) In B+ tree, each leaf node has a pointer that points to the next leaf node i.e. all leaf nodes form a linked list.

The figure 6.163 shows a B tree and the figure 6.164 shows a B+ tree containing the same data.

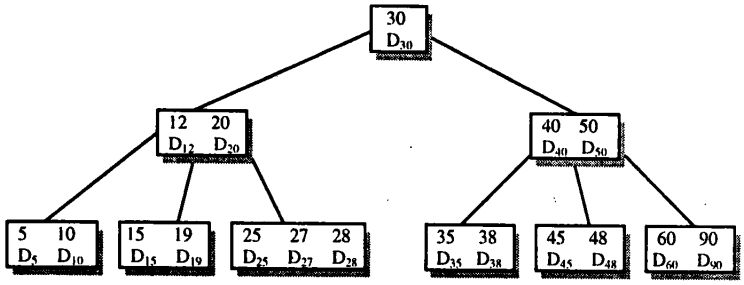


Figure 6.163

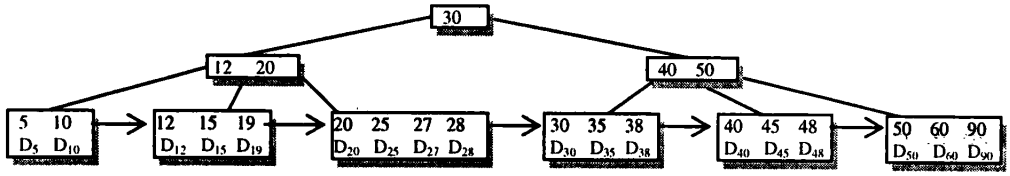


Figure 6.164

The alphabet 'D' with subscript shown under the key value represents the data item. While discussing B tree we had not shown this data item in the figures so that the figures remain small.

In B+ tree, the internal nodes contain only keys and pointers to child nodes, while the leaf nodes contain keys, data items and pointer to next leaf node. The internal nodes are used as index for searching data and so they are also called index nodes. The leaf nodes contain data items and they are also called data nodes. The index nodes form an index set while the data nodes form a sequence set. All the leaf nodes form a linked list and this feature of B+ tree helps in sequential access i.e. we can search for a key and then access all the keys following it in a sequential manner. Traversing all the leaves from left to right gives us all the data in ascending order. So both random and sequential accesses are simultaneously possible in B+ tree.

6.22.1 Searching

In B tree our search terminated when we found the key, but this will not be the case in B+ tree. In a B+ tree, it is possible that a key is present in the internal node but is not present in the leaf node. This happens because when any data item is deleted from the leaf node, the corresponding key is not deleted from the internal node. So presence of a key in an internal node does not indicate that the corresponding data item will be present in the leaf node. Hence the searching process will not stop if we find a key in an internal node but it will continue till we find the key in the leaf node. The data items are stored only in the leaf nodes, so we have to go to the leaf nodes to access the data.

Suppose we want to search the key 20 in the B+ tree of figure 6.164. Searching will start from the root node so first we look at the node [30], and since 20 < 30, we'll move to left child which is [12, 20]. The key value is equal to 20 so we'll move to the right child which is the leaf node and there we find the key 20 with its data item.

B+ tree supports efficient range queries i.e. we can access all data in a given range. For this we need to search the starting key of the range and then sequentially traverse the leaf nodes till we get the end key of the range.

6.21.2 Insertion

First a search is performed and if the key is not present in the leaf node then we can have two cases depending on whether the leaf node has maximum keys or not.

If the leaf node has less than maximum keys, then key and data are simply inserted in the leaf node in ordered manner and the index set is not changed.

If the leaf node has maximum keys, then we will have to split the leaf node. The splitting of a leaf node is slightly different from splitting of a node in a B tree. A new leaf node is allocated and inserted in the sequence set(linked list of leaf nodes) after the old node. All the keys smaller than the median key remain in the old leaf node, all the keys greater than equal to the median key are moved to the new node, the corresponding data items are also moved. The median key becomes the first key of the new node and this key(without data item) is copied(not moved) to the parent node which is an internal node. So now this median key is present both in the leaf node and in the internal node which is the parent of the leaf node.

Splitting of a leaf node
keys < median remain in old leaf node
keys >= median go to new leaf node
Median key is copied to parent node.

If after splitting of leaf node, the parent becomes full then again a split has to be done. The splitting of an internal node is similar to that of splitting of a node in a B tree. When an internal node is split the median key is moved to the parent node.

Splitting of an internal node
keys < median remain in old leaf node
keys > median go to new leaf node
Median key is moved to parent node.

This splitting continues till we get a non full parent node. If root node is split then a new root node has to be allocated.

Suppose we have to insert data with keys 42 and 24 in B+ tree of figure 6.164.

The key 42 can be simply inserted in the leaf node [40, 45, 48]. After inserting 24 in the tree, we get an overflow leaf node [20, 24, 25, 27, 28] which needs to be splitted. A new leaf node is allocated and keys 25, 27, 28 with data items are moved to this node. The median key 25 is copied to the parent node and is present in the leaf node also.

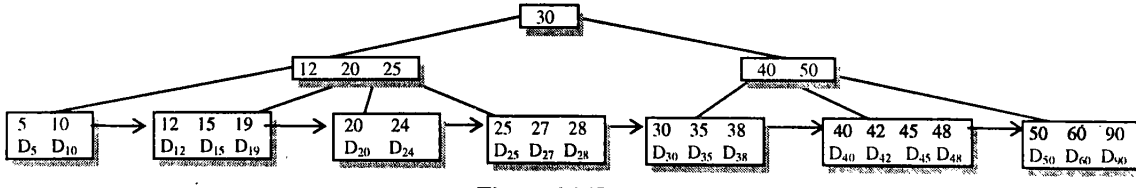


Figure 6.165

6.21.3 Deletion

First a search is performed and if the key is present in the leaf, then we can have two cases depending on whether the leaf node has minimum keys or more than that.

If the leaf node has more than minimum elements then we can simply delete the key and its data item, and move other elements of the leaf node if required. In this case, the index set is not changed i.e. if the key is present in any internal node also then it is not deleted from there. This is because the key still serves as a separator key between its left and right children.

If the key is present in a leaf which has minimum number of nodes then we have two cases-

(A) If any one of the siblings has more than minimum nodes then a key is borrowed from it and the separator key in the parent node is updated accordingly.

If we borrow from left sibling then, then rightmost key(with data item) of left sibling is moved to the underflow node. Now this new leftmost key in the underflow node becomes the new separator key.

If we borrow from right sibling then, then leftmost key(with data item) of right sibling is moved to the underflow node. Now the key which is leftmost in right sibling becomes the new separator key.

(B) If both siblings have minimum nodes then we need to merge the underflow leaf node with its sibling. This is done by moving the keys(with data) of underflow leaf node to the sibling node and deleting the underflow leaf node. The separator key of the underflow node and its sibling is deleted from the parent node, and the corresponding child pointer in parent node is also removed.

The merging of leaf nodes may result in an underflow parent node which is an internal node. For internal nodes borrowing and merging is performed in same manner as in B tree.

(i) Delete data with keys 12, 38, 40, 50 from B+ tree of figure 6.164.

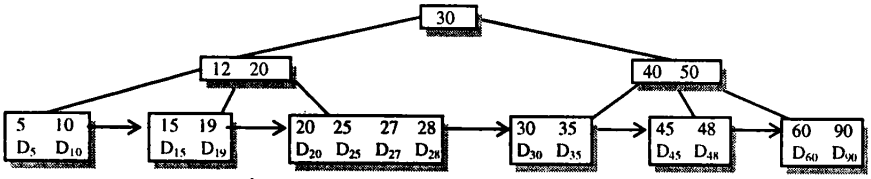


Figure 6.166

(ii) Delete 15 from B+ tree of figure 6.166(borrow from right sibling).

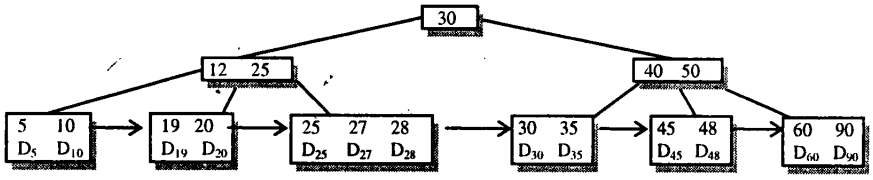


Figure 6.167

(iii) Delete 48 from B+ tree of figure 6.167.

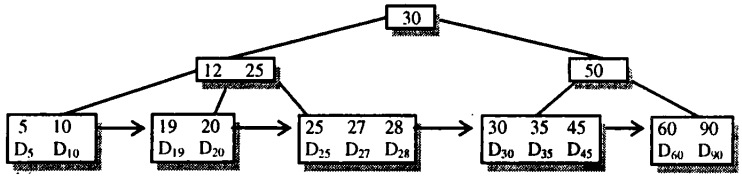


Figure 6.168

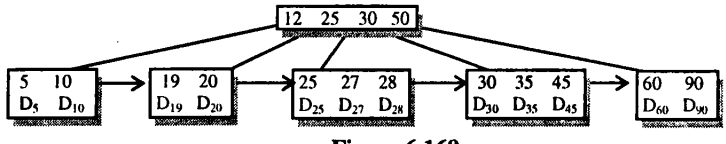


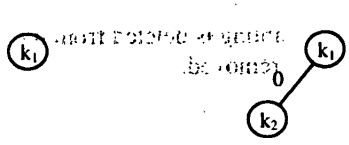
Figure 6.169

6.22 Digital Search Trees

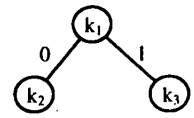
Digital search tree is a binary tree in which a key and data pair is stored in every node, and the position of keys is determined by their binary representation.

The procedure of insertion and searching is similar to that of binary search tree, but with a small difference. In binary search tree, the decision to move to left or right subtree was made by comparing the given key with the key in the current node, while here this decision is made by a bit in the key. The bits in the given key are scanned from left to right and when we have 0 bit we move to the left subtree and when we have a 1 bit we move to the right subtree.

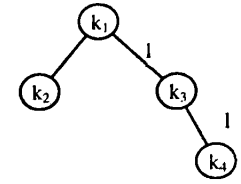
Let us insert some keys into an initially empty digital search tree.



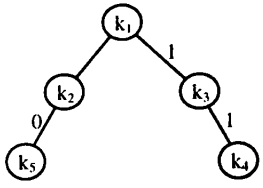
Insert $k_1 = 0110101$ Insert $k_2 = 0010001$



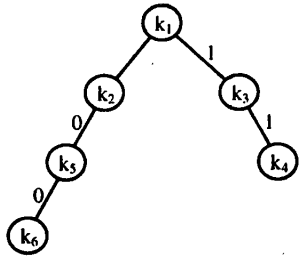
Insert $k_3 = 1010011$



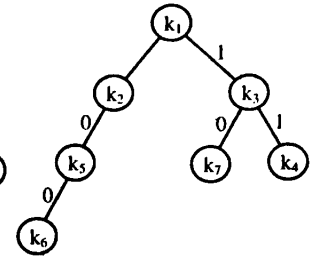
Insert $k_4 = 1100101$



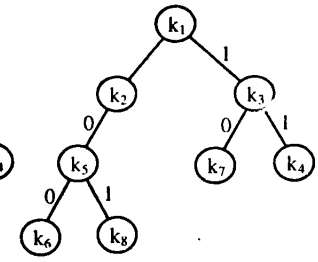
Insert $k_5 = 0011011$



Insert $k_6 = 0001011$



Insert $k_7 = 1001010$



Insert $k_8 = 0011001$

The first key k_1 is to be inserted in an empty tree so it becomes the root of the tree. The next key to be inserted is $k_2 = 0010001$, since this key is not equal to the key in the root, we move down the tree. The first bit from left is 0 so we have to move left and since the left pointer is NULL, we allocate a new node which becomes the left child of root node and insert the key k_2 in that node.

The next key to be inserted is $k_3 = 1010011$, since this key is not equal to the key in the root, we move down the tree. The first bit from left is 1 so we have to move right and since the right pointer is NULL, we allocate a new node which becomes right child of root node and insert the key k_3 in that node.

The next key to be inserted is $k_4 = 1100101$, since this key is not equal to the key in the root, we move down the tree. First bit from left is 1, we move to the right child. The key in right child is not equal to the given key, so we examine second bit in the key k_4 which is 1. Now again we have move to the right and in this case right pointer is NULL so we allocate a new node and insert the key there. Similarly other keys are also inserted.

For searching a key, we proceed down the tree in similar manner. If at any point, the search key is equal to the key in current node, the search is successful. Reaching a NULL pointer implies that key is not present in the tree.

Deletion in DST is much simpler than in BST. If the key to be deleted is in a leaf node, then we can simply remove the leaf node by replacing it with NULL pointer. If the key to be deleted is in a non leaf node, then that key can be replaced with a key from any leaf node in any of its subtree and after that the particular leaf node may be deleted. For example suppose we want to delete key k_2 from last figure. This key can be replaced by any of the keys k_6 or k_8 and then the leaf node may be deleted.

All the above operations are performed in O(h) time where h is the height of the tree. The maximum height of a digital search tree can be p+1 where p is the numbers of bits in the key and so this tree remains balanced.

Exercise

1. Draw all possible non similar binary trees having (i) 3 nodes (ii) 4 nodes.
2. Draw all possible binary trees of 3 nodes having preorder XYZ.
3. Draw all possible binary search trees of 3 nodes having key values 1, 2, 3.
4. Construct a BST by inserting the following data sequentially.
   45 32 70 67 21 85 92 40

5. If we construct a binary search tree by inserting the following data sequentially, then what is the height of the tree formed.

71 32 12 82 45 91 38 70 40 61

If the binary search tree is constructed by inserting this data in sorted order, then what will be the height of that tree.

6. The preorder traversal of a binary search tree T is 23 12 11 9 6 45 32 67 56. What are the inorder and postorder traversals of the tree T.

7. Show a binary tree for which preorder and inorder traversals are same.

8. Show a binary tree for which postorder and inorder traversals are same.

9. Construct binary trees from inorder and preorder traversals.

(i) Inorder : 12 31 10 45 66 Preorder : 12 31 10 45 66

(ii) Inorder : 35 26 93 21 68 Preorder : 68 21 93 26 35

(iii) Inorder : 16 22 31 15 46 77 19 Preorder : 15 22 16 31 77 46 19

(iv) Inorder 11 12 23 24 25 32 43 46 54 65 Preorder : 32 23 11 12 24 25 43 54 46 65

10. Construct binary trees from inorder and postorder traversals.

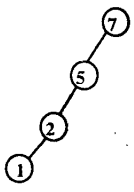
(i) Inorder : 12 31 10 45 66 Postorder : 12 31 10 45 66

(ii) Inorder : 35 26 93 21 68 Postorder : 68 21 93 26 35

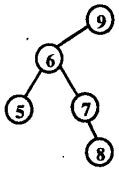
(iii) Inorder : 20 19 24 8 11 13 6 Postorder : 20 24 19 11 6 13 8

(iv) Inorder : 4 5 6 11 19 23 43 50 54 98 Postorder : 4 6 5 19 11 50 98 54 43 23

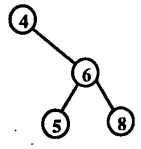
11. For the following binary search trees, show the possible sequences in which the data was entered in these trees.



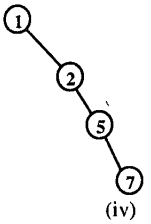
(i)



(ii)



(iii)



(iv)

12. Suppose a binary search tree is constructed by inserting the keys 1, 2, 3, 4.....n in any order.

(a) If there are x nodes in right subtree of root, which key was inserted in the beginning.

(b) If there are y nodes in left subtree of root, which key was inserted in the beginning.

13. We know that preorder and postorder traversals can't uniquely define a binary tree. Show example of binary trees that have same preorder and postorder traversals.

14. Construct a binary search tree whose preorder traversal is-

67 34 12 45 38 60 80 78 95 90

15. Construct a binary search tree whose postorder traversal is-

10 11 40 48 44 32 65 73 88 77 72 56

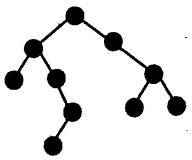
16. Construct a full binary tree whose preorder is -

F B G I C K L

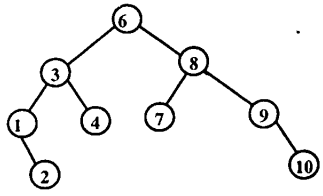
17. Write a function that returns the size of a binary tree i.e. the total number of nodes in the tree.

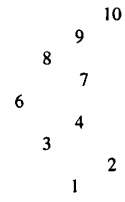
18. Write a function that returns the total number of leaf nodes in a binary tree and displays the info part of each leaf node.

19. Write a function to find the length of shortest path from root to a leaf node, this length is also known as minimum height of the binary tree. For example the minimum height of this tree is 3.

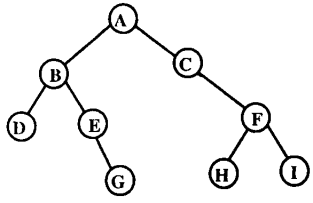


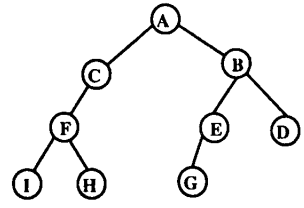
20. Write a function to destroy all the nodes of a binary tree.
21. Write a function to display all the ancestors of a node in a binary tree.
22 Write a function that displays all root to leaf paths in a binary tree.
23. Write a function to display a binary tree from left to right.



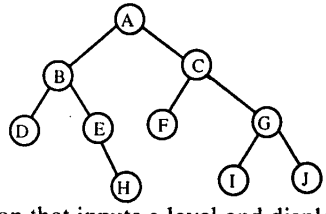


24. Write a function to create a copy of binary tree.
25. Two binary trees are similar if their structure is same. Write a function to check whether two binary trees are similar or not.
26. Write a function to check whether two Binary trees are identical or not. Two binary trees are identical or copies if structure as well as data is same.
27. Write a function to swap right and left children of a binary tree i.e. all right children become left children and vice versa. The new tree is the mirror image of the original tree.





28. Write a function to find whether two binary trees are mirror image of each other or not.
29. Write a function to check whether a binary tree is binary search tree or not.
30. Write a recursive function that inputs a level number of a binary tree and returns the number of nodes at that level.
31. Width of a binary tree is the number of nodes on the level that has maximum nodes. For example width of the following binary tree is 4. Write a function that returns the width of a binary tree.



32. Write a function that inputs a level and displays nodes on that level from right to left.
33. Write a function to traverse a tree in spiral or zigzag order. The spiral traversal of tree in exercise 31 is A C B D E F G J I H.
34. Draw an expression tree for the following algebraic expression and write the prefix and postfix forms of the expression by traversing the expression tree in and preorder and postorder.

(a + b / c) – (d + e * f)

35. The level order traversal of a max heap is 50 40 30 25 16 23 20. What will be the level order traversal after inserting the elements 28, 43, 11.
36. Generate Huffman code for the letters a, b, c, d, e, f having frequencies 16, 8, 4, 2, 1, 1.
37. The following function tries to find out whether a BST is AVL tree or not. Trace and find out whether it gives correct output or not. Write the modified function if required.

```
int isAVL(struct node *ptr)
{
        int h_l,h_r,diff;
        if(ptr == NULL)
                return 1;
        h_l = height(ptr->lchild);
        h_r = height(ptr->rchild);
        diff = h_l>h_r ? h_l-h_r : h_r-h_l;
        if(diff<=1)
                return 1;
        return 0;
}
```

38. Construct an AVL tree by inserting the following values sequentially.

23 34 12 11 6 2 45 4 25 24

39. Construct a B tree of order 5 by inserting the following key values sequentially.

35 63 24 10 12 39 89 72 11 8 4 18 78 14 80 70 21

Graphs

A graph G = (V, E) is a collection of sets V and E where V is the collection of vertices and E is the collection of edges. An edge is a line or arc connecting two vertices and it is denoted by a pair (i, j) where i, j belong to the set of vertices V. A graph can be of two types - Undirected graph or Directed graph.

7.1 Undirected Graph

A graph, which has unordered pair of vertices, is called undirected graph. If there is an edge between vertices u and v then it can be represented as either (u, v) or (v, u).

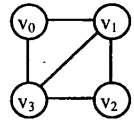


Figure 7.1 Undirected Graph

$V(G) = \{ v_0, v_1, v_2, v_3 \}$
$E(G) = \{ (v_0, v_1), (v_0,v_3), (v_1,v_2), (v_1,v_3), (v_2,v_3) \}$

This graph is undirected; it has 4 vertices and 5 edges.

7.2 Directed Graph

A directed graph or digraph is a graph which has ordered pair of vertices (u,v) where u is the tail and v is the head of the edge. In this type of graph, a direction is associated with each edge i.e. (u,v) and (v,u) represent different edges.

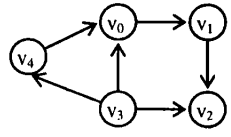


Figure 7.2 Directed Graph

$V(G) = \{ v_0, v_1, v_2, v_3, v_4 \}$
$E(G) = \{ (v_0,v_1), (v_1,v_2), (v_3,v_2),(v_3,v_0),(v_3,v_4),(v_4,v_0) \}$

This graph is directed; it has 5 vertices and 6 edges.

7.3 Graph Terminology

The definitions and terms associated with graph data structures are explained with the help of example graphs shown in figure 7.3.

Weighted graph - A graph is weighted if its edges have been assigned some non negative value as weight. A weighted graph is also known as network. Graph G9 is a weighted graph. The weight on the edge may represent cost, length or distance associated with the edge.

Subgraph - A graph H is said to be a subgraph of another graph G, iff the vertex set of H is subset of vertex set of G and edge set of H is subset of edge set of G.

Adjacency - Adjacency is a relation between two vertices of a graph. A vertex v is adjacent to another vertex u if there is an edge from vertex u to vertex v i.e. edge (u,v) $\in$ E.

In an undirected graph if we have an edge (u,v), it means that there is an edge from u to v and also an edge from v to u. So the adjacency relation is symmetric for undirected graphs, i.e. if (u,v) is an edge then u is adjacent to v and v is adjacent to u. For example in G2, vertex v_0 is adjacent to v_3 and v_3 is adjacent to v_0, vertex v_0 is not adjacent to v_2 since there is no edge between them.

In a digraph if (u,v) is an edge, then v is adjacent *to* u but u is not adjacent to v since there is no edge from v to u. The vertex u is said to be adjacent *from* v. For example in G5, v_1 is adjacent to v_0 but v_0 is not adjacent to v_1. The vertex v_0 is adjacent from v_1.

Incidence - Incidence is a relation between a vertex and an edge of a graph. In an undirected graph the edge (u,v) is incident on vertices u and v. For example in G2 edge(v_0,v_3) is incident on vertices v_0 and v_3.

In a digraph, the edge (u,v) is incident from vertex u and is incident to vertex v. For example in G5, the edge(v_0,v_1) is incident from vertex v_0 and incident to vertex v_1.

Path - A path from vertex u_1 to vertex u_n is a sequence of vertices u_1, u_2, u_3,.....,u_{n-1}, u_n such that u_2 is adjacent to u_1, u_3 is adjacent to u_2, ... , u_n is adjacent to u_{n-1}. In other words we can say that (u_1,u_2), (u_2,u_3) .....(u_{n-1},u_n) are all edges or (u_i , u_{i+1}) $\in$ E for i=1,2,3.....n-1. For example in digraph G5, v_3–v_4–v_0–v_1–v_2 is a path, while v_3–v_2–v_1 is not a path since v_1 is not adjacent to v_2. In undirected graph G12, v_5–v_2–v_3 and v_0–v_1–v_4–v_3–v_1 are examples of path.

Length of a path - The length of a path is the total number of edges included in the path. For a path with vertices u_1, u_2, u_3,.....,u_{n-1}, u_n, the length of path is n-1. For example the length of path v_3–v_4–v_0–v_1–v_2 in G5 is 4.

Reachable - If there is a path P from vertex u to vertex v, then vertex v is said to be reachable from vertex u via path P. For example in digraph G5, vertex v_2 is reachable from vertex v_4 via path v_4–v_0–v_1–v_2 while vertex v_3 is not reachable from vertex v_4 as there is no path from v_4 to v_3.

Simple path - Simple path is a path in which all the vertices are distinct. For example in graph G12, path v_0–v_1–v_3–v_4–v_6 is a simple path while path v_0–v_1–v_3–v_4–v_6–v_3–v_2 is not a simple path because vertex v_3 is repeated.

Cycle - In a digraph, a path u_1, u_2, ... , u_{n-1},u_n is called a cycle if it has at least two vertices and the first and last vertices are same i.e. $u_1 = u_n$. In graph G7, path v_0–v_2–v_1–v_0 is a cycle and in graph G9 path v_0–v_1–v_0 is a cycle.

In an undirected graph, a path u_1, u_2, ..., u_{n-1}, u_n is called a cycle if it has at least three vertices and the first and last vertices are same i.e. $u_1 = u_n$. In undirected graph if (u,v) is an edge then u–v–u should not be considered a cycle since (u,v) and (v,u) represent the same edge. So for a path to be a cycle in an undirected graph there should be at least three vertices. For example in graph G12, v_1–v_4–v_6–v_3–v_1 is a cycle of length 4, path v_6-v_4-v_3-v_6 is a cycle of length 3.

Simple Cycle - A cycle u_1, u_2, u_3,.....,u_{n-1}, u_n is simple if the vertices u_2, u_3, ... ,u_{n-1}, u_n are distinct. For example in graph G12, v_1–v_4–v_3–v_2–v_0–v_1 is a simple cycle but v_1–v_4–v_3–v_6–v_2–v_3–v_1 is not a simple cycle because vertex v_3 is repeated.

Cyclic graph - A graph that has one or more cycles is called a cyclic graph. Graphs G1, G3, G7, G9, G11 and G12 are examples of cyclic graphs.

Acyclic graph - A graph that has no cycle is called an acyclic graph. Graphs G2, G4, G5, G6, G8 and G10 are examples of acyclic graphs.

DAG - A directed acyclic graph is named DAG after its acronym. Graph G5 is an example of a dag.

Degree - In an undirected graph, the degree of a vertex is the number of edges incident on it. In graph G12, degree of vertex v_0 is 2, degree of v_1 is 3, degree of v_2 is 4, degree of v_3 is 4, degree of v_4 is 3, degree of v_5 is 2 and degree of v_6 is 4. In a digraph, each vertex has an indegree and an outdegree. The degree of a vertex in a digraph is the sum of its indegree and outdegree.

Indegree - The indegree of a vertex v is the number of edges entering the vertex v, or in other words the number of edges incident to vertex v. In graph G8, the indegree of vertices v_0, v_1, v_3 and v_6 are 0, 2, 6 and 1 respectively.

Outdegree - The outdegree of vertex v is the number of edges leaving the vertex v or in other words the number of edges which are incident from v. In graph G8, outdegrees of vertices v_0, v_1, v_3, v_5, and v_6 are 3, 1, 0, 3, and 2 respectively.

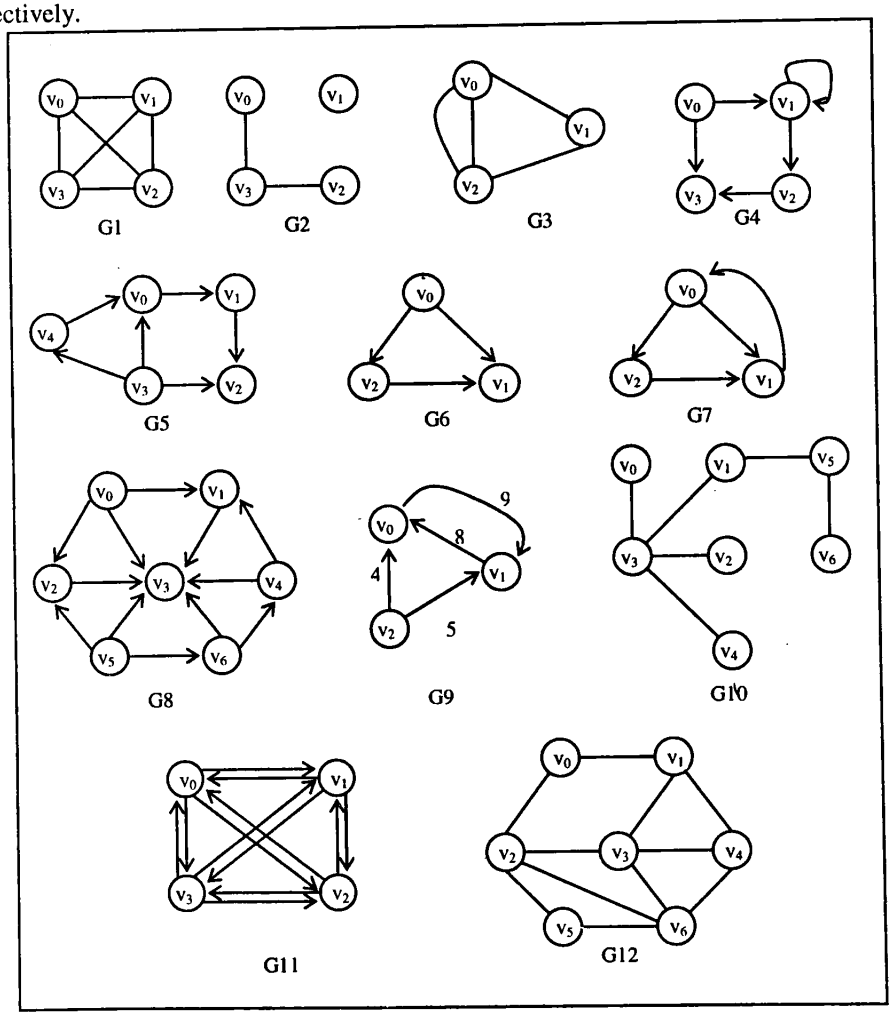


Figure 7.3

Source - A vertex, which has no incoming edges, but has outgoing edges, is called a source. The indegree of a source is zero. In graph G8, vertices v_0 and v_5 are sources.

Sink - A vertex, which has no outgoing edges but has incoming edges, is called a sink. The outdegree of a sink is zero. In graph G8, vertex v_3 is a sink.

Pendant vertex - A vertex in a digraph is said to be pendant if its indegree is equal to 1 and outdegree is equal to 0.

Isolated vertex - If the degree of a vertex is 0, then it is called an isolated vertex. In graph G2, vertex v_1 is an isolated vertex.

Successor and predecessor - In a digraph, if a vertex v is adjacent to vertex u, then v is said to be the successor of u, and u is said to be the predecessor of v. In graph G8, v_0 is predecessor of v_1 while v_1 is successor of v_0.

Maximum edges in a graph - If n is the total number of vertices in a graph, then an undirected graph can have maximum n(n-1)/2 edges and a digraph can have maximum n(n-1) edges. For example an undirected graph with 3 vertices can have maximum 3 edges, and an undirected graph with 4 vertices can have maximum 6 edges. A digraph with 3 vertices can have maximum 6 edges and a digraph with 4 vertices can have maximum 12 edges.

Complete graph - A graph is complete if any vertex in the graph is adjacent to all the vertices of the graph or we can say that there is an edge between any pair of vertices in the graph. A complete graph contains maximum number of edges, so an undirected complete graph with n vertices will have n(n-1)/2 edges and a directed complete graph with n vertices will have n(n-1) edges. Graph G1 is a complete undirected graph and graph G11 is a complete directed graph.

Multiple edges - If there is more than one edge between a pair of vertices then the edges are known as multiple edges or parallel edges. In graph G3, there are multiple edges between vertices v_0 and v_2.

Loop - An edge is called loop or self edge if it starts and ends on the same vertex. Graph G4 has a loop at vertex v_1.

Multigraph - A graph which contains loop or multiple edges is known as multigraph. Graphs G3 and G4 are multigraphs.

Simple graph - A graph which does not have loop or multiple edges is known as simple graph.

Regular graph - A graph is regular if every vertex is adjacent to the same number of vertices. Graph G1 is regular since every vertex is adjacent to 3 vertices.

Planar graph - A graph is called planar if it can be drawn in a plane without any two edges intersecting. Graph G1 is not a planar graph, while graphs G2, G3, G4 are planar graphs.

Null graph - A graph which has only isolated vertices is called null graph.

7.4 Connectivity in Undirected Graph

7.4.1 Connected Graph

An undirected graph is connected if there is a path from any vertex to any other vertex, or any vertex is reachable from any other vertex. A connected graph of n vertices has at least n-1 edges.The following are some examples of connected graphs.

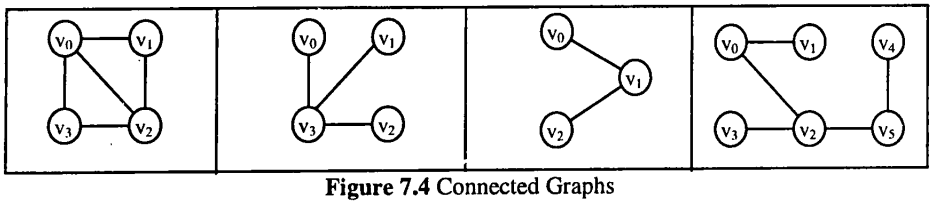


Figure 7.4 Connected Graphs

The three graphs given next are not connected graphs.

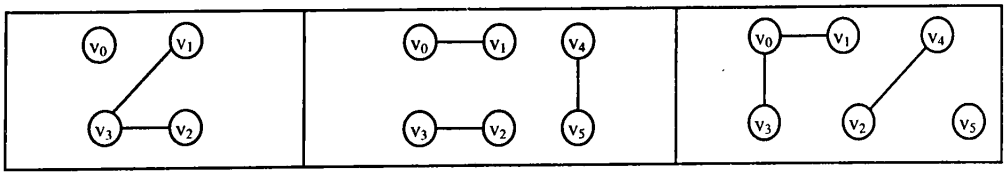


Figure 7.5 Graphs which are not connected

7.4.2 Connected Components

An undirected graph which is not connected may have different parts of the graph which are connected. These parts are called connected components.

A connected component of an undirected graph is a subgraph in which all vertices are connected by paths, and it is not possible to add any other vertex to it while retaining its connectivity property. This way we can define a connected component as a maximal connected subgraph. The figure 7.6 shows three graphs with their connected components.

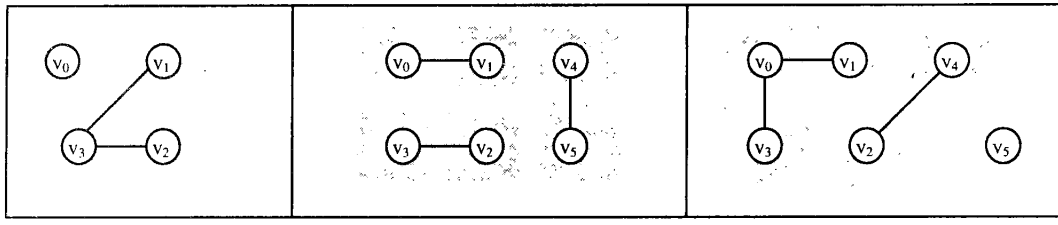


Figure 7.6 Connected Components

If a graph is connected, it has only one connected component which consists of the whole graph.

7.4.3 Bridge

If on removing an edge from a connected graph, the graph becomes disconnected then that edge is called a bridge. Consider the following graph, we will remove all the edges one by one from it and see if the graph becomes disconnected.

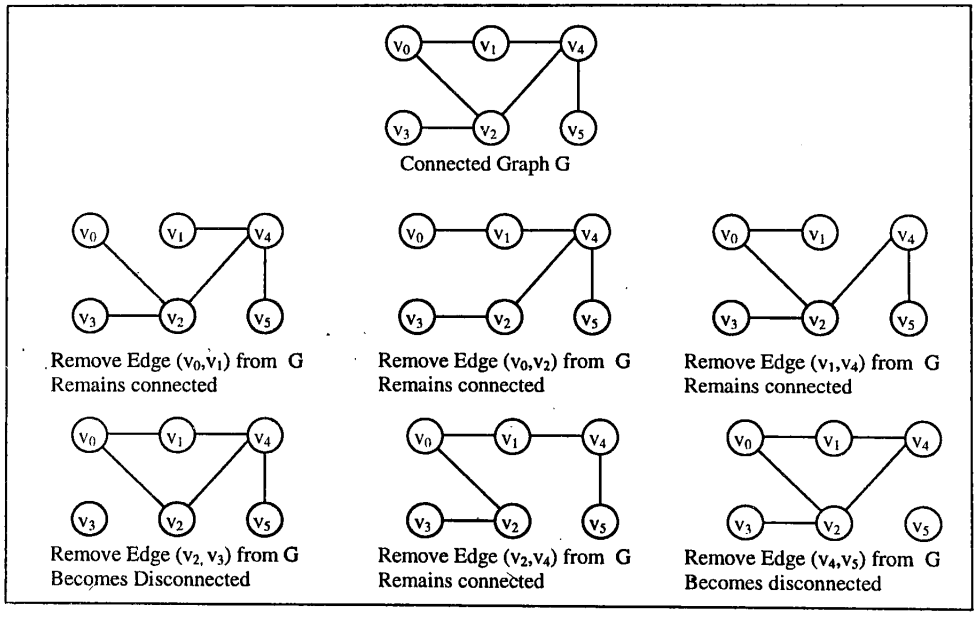


Figure 7.7

From the figure 7.7, we find that removal of edge (v_2, v_3) and removal of edge (v_4, v_5) makes the graph disconnected so these edges are bridges.

7.4.4 Articulation point

If on removing a vertex from a connected graph, the graph becomes disconnected then that vertex is called the articulation point or a cut vertex. Consider the graph in figure 7.8, we will remove all the vertices one by one from it and see if the graph becomes disconnected.

From the figure 7.8, we find that removal of vertex v_2 and removal of vertex v_4 makes the graph disconnected so these are the articulation points of this graph.

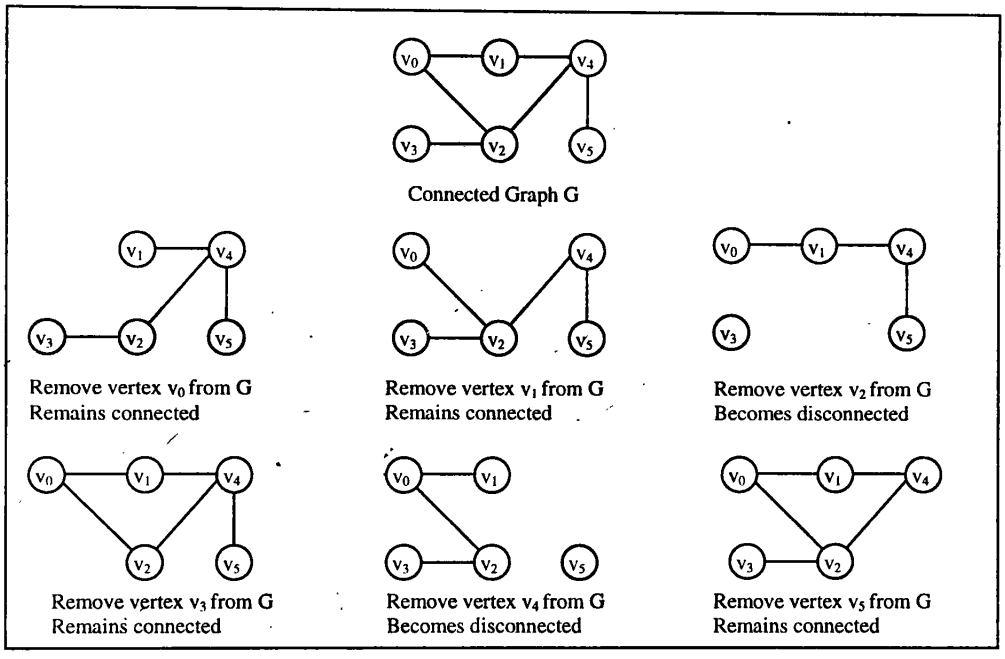


Figure 7.8

Here are some more examples of graphs with articulation points.

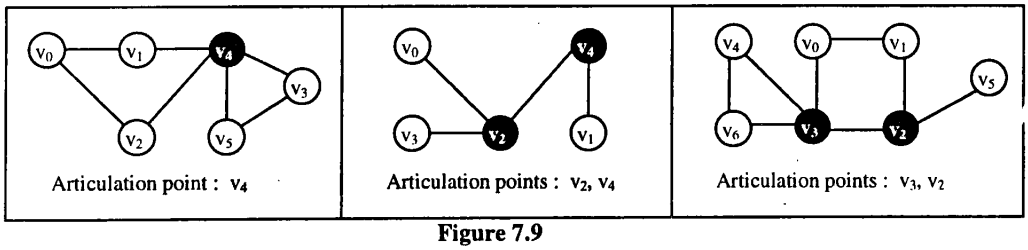


Figure 7.9

7.4.5 Biconnected graph

A connected graph with no articulation points is called a biconnected graph. Thus a biconnected graph is a graph which is connected and it does not have any vertex whose removal can disconnect the graph. Some examples of biconnected graphs are-

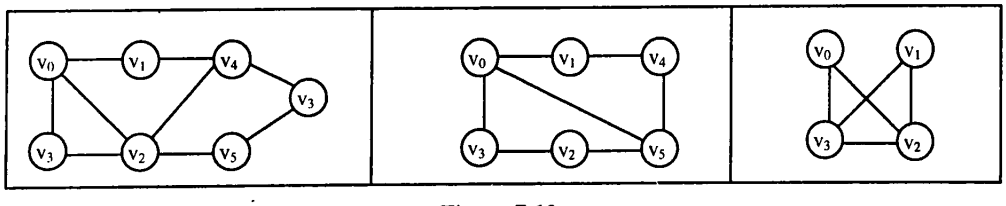


Figure 7.10

7.4.6 Biconnected components

A biconnected component is a maximal biconnected subgraph. It is a maximal set of edges such that any two edges of this set lie on a common simple cycle.

In the figure 7.11, the black vertices indicate articulation points, bold edges indicate bridges and edges in the separate shaded regions indicate biconnected components.

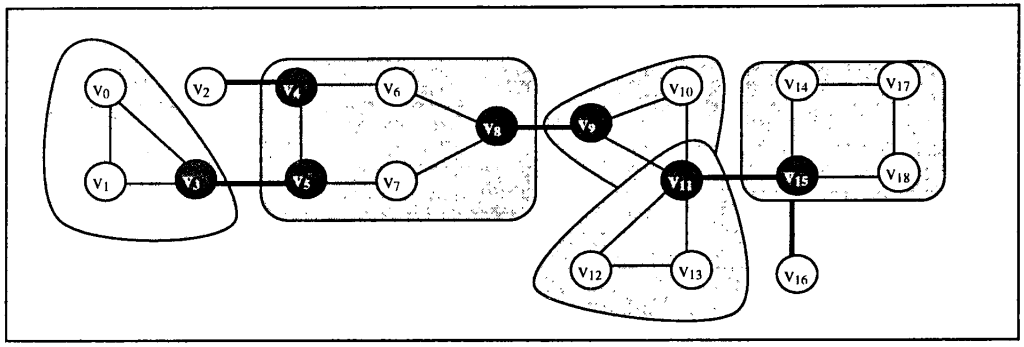


Figure 7.11

7.5 Connectivity in Directed Graphs

7.5.1 Strongly connected Graph

A digraph is strongly connected if there is a directed path from any vertex of graph to any other vertex. We can also say that a digraph is strongly connected if for any pair of vertices u and v, there is a path from u to v and also a path from v to u. The three graphs given next are the examples of strongly connected graphs.

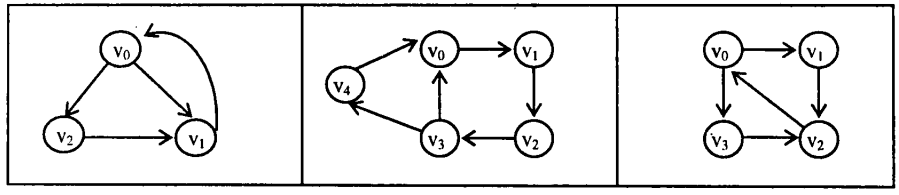


Figure 7.12 Strongly connected Graphs

Here are some graphs which are not strongly connected.

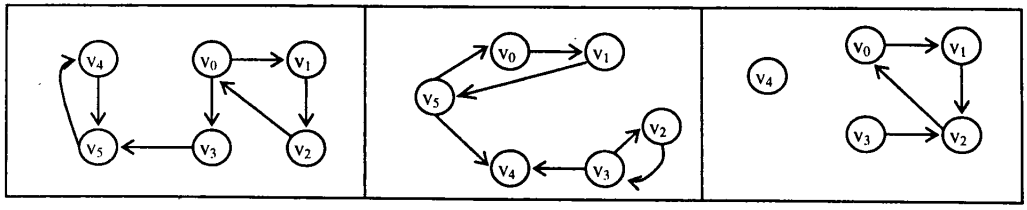


Figure 7.13 Graphs which are not strongly connected

7.5.2 Strongly connected components

A digraph which is not strongly connected may have different parts of the graph which are strongly connected. These parts are called strongly connected components. A strongly connected component of a graph is a

maximal strongly connected subgraph. The following figure shows some graphs with their strongly connected components.

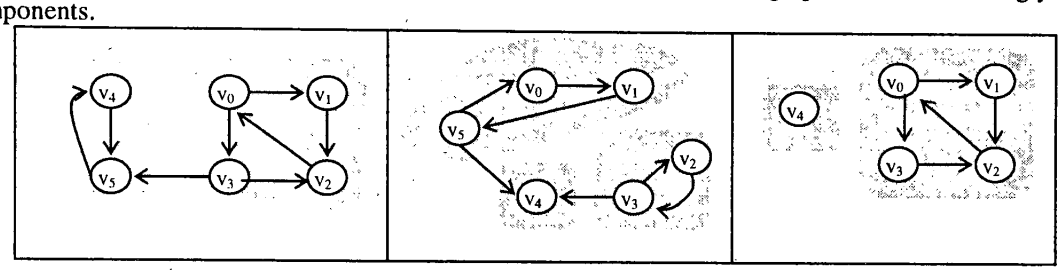


Figure 7.14 Strongly connected components

7.5.3 Weakly connected

A digraph is called weakly connected or unilaterally connected if for any pair of vertices u and v, there is a path from u to v or a path from v to u or both. From a digraph, if we remove the directions and the resulting undirected graph is connected then that digraph is weakly connected. In the following figure, the first graph is weakly connected while the second one is not.

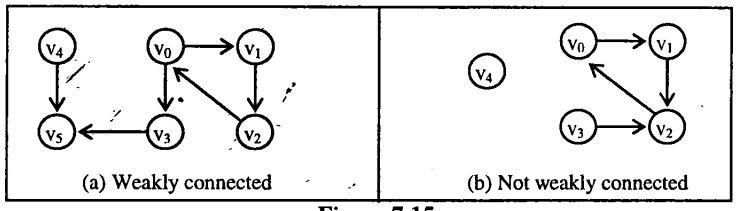


Figure 7.15

7.6 Tree

An undirected connected graph T is called tree if there are no cycles in it. There is *exactly* one simple path between any two vertices u and v of T. If there is more than one path between any two vertices, then it would mean that there is a cycle in the graph and if there is no path between any two vertices then it would mean that graph is not connected. So according to the definition of tree, there will be exactly one simple path between any pair of vertices of the tree. A tree with n vertices will have exactly n-1 edges. The following are some examples of trees.

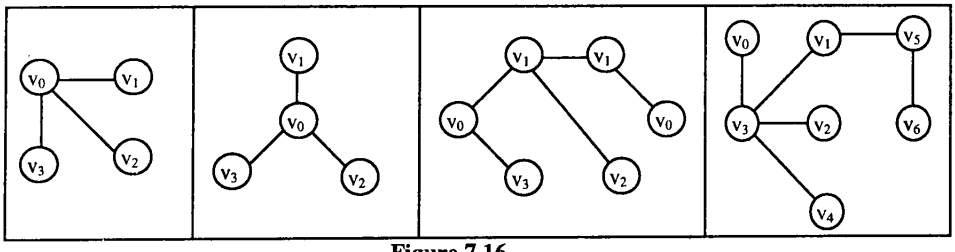


Figure 7.16

The following examples are graphs which are not trees.

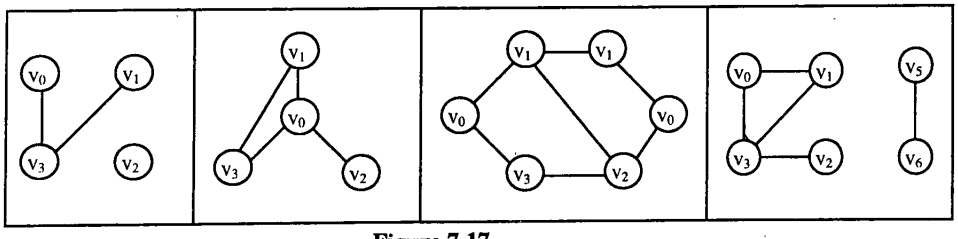


Figure 7.17

In figure 7.17, the first graph is not a tree as it is not connected, the next two graphs are not trees as they are cyclic, and the last graph is not a tree as it is not connected and is cyclic.

If any edge is removed from a tree, then the graph will not remain connected, i.e. all edges in a tree are bridges. If any edge is added to the tree then a simple cycle is formed.

7.7 Forest

A forest is a disjoint union of trees. In a forest there is *at most* one path between any two vertices, this means that there is either no path or a single path between any two vertices.

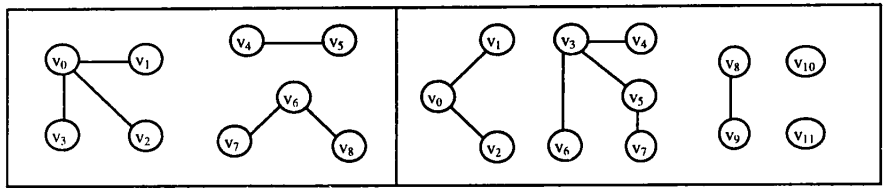


Figure 7.18

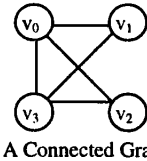
The first forest in the figure 7.18 consists of 3 trees, and the second forest consists of 5 trees.

7.8 Spanning tree

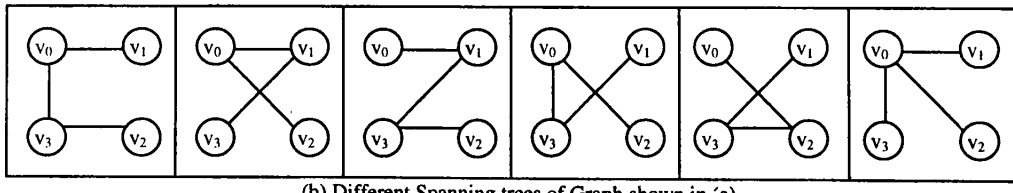
A subgraph T of a connected graph G, which contains all the vertices of G and is a tree is called a spanning tree of G. It is called spanning tree because it spans over all vertices of graph G.

So a spanning tree of a graph G is a subgraph that includes all vertices of G and some (or all) edges of G, such that all the vertices are connected and there are no cycles.

Spanning tree of a graph is not unique; there can be more than one spanning trees of a graph. The figure 7.19 shows a connected graph and its six different spanning trees.



(a) A Connected Graph



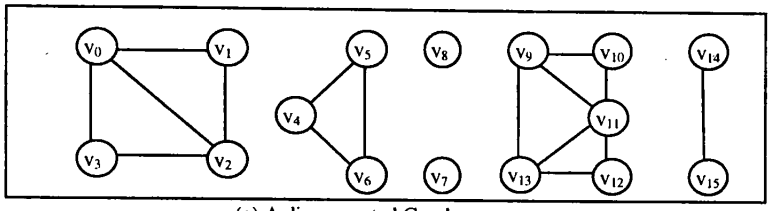
(b) Different Spanning trees of Graph shown in (a)

Figure 7.19

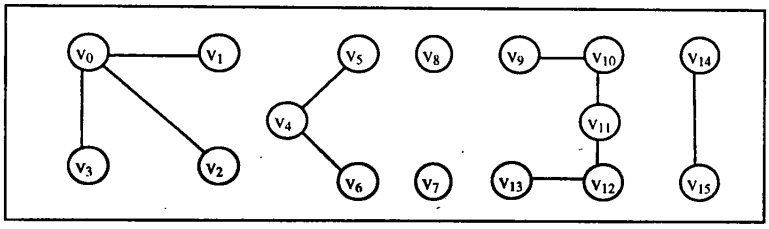
If a graph is connected then it will always have a spanning tree. If a graph G is not connected then there will be no spanning tree of G.

7.9 Spanning Forest

A spanning forest is a subgraph that consists of a spanning tree for each connected component of a graph. The following figure shows a graph and its spanning forest. There can be many other spanning forests for this graph.



(a) A disconnected Graph



(b) A Spanning forest of Graph shown in (a)

Figure 7.20

7.10 Representation of Graph

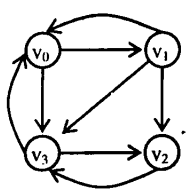
We have mainly two parts in a graph viz. vertices and edges, and we have to design a data structure keeping these parts in mind. There are two ways of representing a graph, one is the sequential representation(adjacency matrix) and other is the linked representation(adjacency list).

7.10.1 Adjacency Matrix

Adjacency matrix is a matrix that maintains the information of adjacent vertices. In other words, we can say that this matrix tells us whether a vertex is adjacent to any other vertex or not. Suppose there are 4 vertices in a graph then first row represents the vertex 1, second row represents the vertex 2 and so on. Similarly first column represents vertex 1, second column represents vertex 2 and so on. The entries of this adjacency matrix are filled using this definition-

$$A(i,j) = \begin{cases} 1 & \text{If there is an edge from vertex i to vertex j} \\ 0 & \text{If there is no edge from vertex i to vertex j} \end{cases}$$

Hence, all the entries of this matrix are either 1 or 0. Let us take a directed graph and write the adjacency matrix for it.



(a) Directed Graph

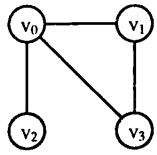
	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	1
v_2	0	0	0	1
v_3	1	0	1	0

(b) Adjacency Matrix for graph (a)

Figure 7.21

Here the matrix entry A(0,1) = 1, which means that there is an edge in the graph from vertex v_0 to vertex v_1. Similarly A(2,0) = 0, which means that there is no edge from vertex v_2 to vertex v_0. In the adjacency matrix of a directed graph, rowsum represents the outdegree and columnsum represents the indegree of that vertex. For example from the above matrix we can see that the rowsum of vertex v_1 is 3 which is its outdegree and columnsun is 1 which is its indegree.

Let us take an undirected graph and write the adjacency matrix for it.



(a) Undirected Graph

	v_0	v_1	v_2	v_3
v_0	0	1	1	1
v_1	1	0	0	1
v_2	1	0	0	0
v_3	1	1	0	0

(b) Adjacency Matrix for graph (a)

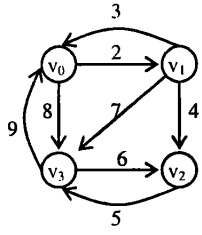
Figure 7.22

In an undirected graph if there is an edge from i to j, then there will also be an edge from j to i, i.e. A(i,j) = A(j,i) for every i and j. Hence the adjacency matrix for an undirected graph will be a symmetric matrix. In an undirected graph, rowsum and columnsum for a vertex are equal and represent the degree of that vertex.

If a graph has some weights on its edges then the elements of adjacency matrix can be defined as-

$$A(i,j) = \begin{cases} \text{Weight on edge} & \text{If there is an edge from vertex i to vertex j.} \\ 0 & \text{Otherwise} \end{cases}$$

Let us take a directed weighted graph and write the weighted adjacency matrix for it.



(a) Weighted Directed Graph

	v_0	v_1	v_2	v_3
v_0	0	2	0	8
v_1	3	0	4	7
v_2	0	0	0	5
v_3	9	0	6	0

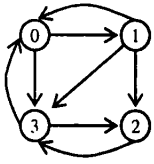
(b) WeightedAdjacency Matrix for graph (a)

Figure 7.23

Here all the non-zero elements of matrix represent the weight on the corresponding edge.

We know that in C, we can represent a matrix by a two dimensional array, where first subscript represents row and second subscript represents column of that matrix.

Suppose we have n vertices in a graph, and these vertices are represented by integers from 0 to n-1. The adjacency matrix of this graph can be maintained with a 2 -mensional integer array `adj[n][n]`.



(a) A Directed Graph

	0	1	2	3
0	0	1	0	1
1	1	0	1	1
2	0	0	0	1
3	1	0	1	0

(b) Adjacency Matrix

	0	1	2	3
0	0	1	0	1
1	1	0	1	1
2	0	0	0	1
3	1	0	1	0

(c) Adjacency Matrix maintained in a 2-d array

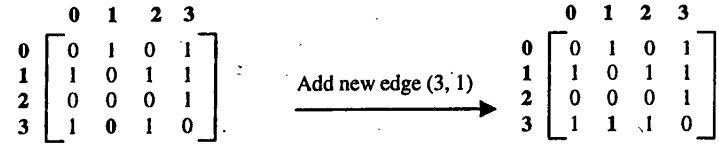
Figure 7.24

This adjacency matrix is maintained in the array `adj[4][4]`. The following program shows how to create and display an adjacency matrix of a graph.

```
/*P7.1 Program for creation of adjacency matrix*/
#include<stdio.h>
#define MAX 100
int adj[MAX][MAX]; /*Adjacency matrix*/
int n;    /*Number of vertices in the graph*/
main()
```

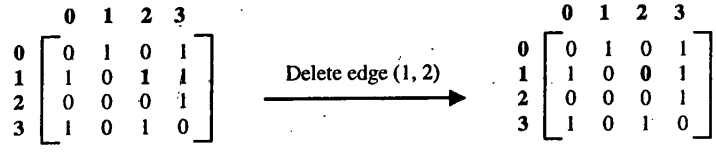
```
{
        int max_edges,i,j,origin,destin;
        int graph_type;
        printf("Enter 1 for undirected graph or 2 for directed graph : ");
        scanf("%d",&graph_type);
        printf("Enter number of vertices : ");
        scanf("%d",&n);
        if(graph_type==1)
                max_edges=n*(n-1)/2;
        else
                max_edges=n*(n-1);
        for(i=1; i<=max_edges; i++)
        {
                printf("Enter edge %d(-1 -1 to quit) : ",i);
                scanf("%d %d",&origin,&destin);
                if((origin==-1) && (destin==-1))
                        break;
                if(origin>=n || destin>=n || origin<0 || destin<0)
                {
                        printf("Invalid vertex!\n");
                        i--;
                }
                else
                {
                        adj[origin][destin]=1;
                        if(graph_type==1) /*if undirected graph*/
                                adj[destin][origin]=1;
                }
        }/*End of for*/
        printf("The adjacency matrix is :\n");
        for(i=0; i<=n-1; i++)
        {
                for(j=0; j<=n-1; j++)
                        printf("%4d",adj[i][j]);
                printf("\n");
        }
}/*End of main()*/
```

Insertion of an edge (i, j) requires changing the value of `adj[i][j]` from 0 to 1.



Initially there was no edge from vertex 3 to vertex 1, so there was 0 in the 4th row 2nd column. After insertion of edge (3, 1), this 0 changes to 1.

Deletion of an edge (i, j) requires changing the value of `adj[i][j]` from 1 to 0.



Initially there exists an edge from vertex 1 to vertex 2, so there is 1 in the 2nd row 3rd column. After deletion of this edge, this 1 changes to 0.

```
/*P7.2 Program for addition and deletion of edges in a directed graph using adjacency
matrix*/
#include<stdio.h>
```

```
#define MAX 100
int adj[MAX][MAX];
int n;
void create_graph();
void display();
void insert_edge(int origin,int destin);
void del_edge(int origin, int destin);
main()
{
        int choice,origin,destin;
        create_graph();
        while(1)
        {
                printf("1.Insert an edge\n");
                printf("2.Delete an edge\n");
                printf("3.Display\n");
                printf("4.Exit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1:
                        printf("Enter an edge to be inserted : ");
                        scanf("%d %d",&origin,&destin);
                        insert_edge(origin,destin);
                        break;
                 case 2:
                        printf("Enter an edge to be deleted : ");
                        scanf("%d %d",&origin,&destin);
                        del_edge(origin,destin);
                        break;
                 case 3:
                        display();
                        break;
                 case 4:
                        exit();
                 default:
                        printf("Wrong choice\n");
                        break;
                 }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
void create_graph()
{
        int i,max_edges,origin,destin;
        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edges=n*(n-1); /*directed graph*/
        for(i=1; i<=max_edges; i++)
        {
             printf("Enter edge %d( -1 -1 ) to quit : ",i);
             scanf("%d %d",&origin,&destin);
             if((origin==-1) && (destin==-1))
                    break;
             if(origin>=n || destin>=n || origin<0 || destin<0)
             {
                    printf("Invalid edge!\n");
                    i--;
             }
             else
                    adj[origin][destin]=1;
        }/*End of for*/
}/*End of create_graph()*/
```

```
void insert_edge(int origin,int destin)
{
        if(origin<0 || origin>=n)
        {
                printf("Origin vertex does not exist\n");
                return;
        }
        if(destin<0 || destin>=n)
        {
                printf("Destination vertex does not exist\n");
                return;
        }
        adj[origin][destin]=1;
}/*End of insert_edge()*/
void del_edge(int origin, int destin)
{
  if(origin<0 || origin>=n || destin<0 || destin>=n || adj[origin][destin]==0)
  {
        printf("This edge does not exist\n");
        return;
  }
  adj[origin][destin] = 0;
}/*End of del_edge()*/
void display()
{
        int i,j;
        for(i=0; i<n; i++)
        {
                for(j=0; j<n; j++)
                        printf("%4d",adj[i][j]);
                printf("\n");
        }
}/*End of display()*/
```

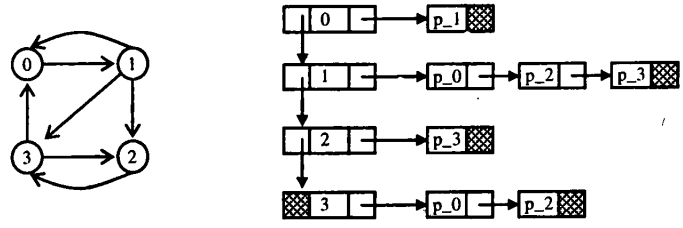
7.10.2 Adjacency List

If the graph is not dense i.e. the number of edges is less, then it is efficient to represent the graph through adjacency list.

In adjacency list representation of graph, we maintain two linked lists. The first linked list is the vertex list that keeps track of all the vertices in the graph and second linked list is the edge list that maintains a list of adjacent vertices for each vertex. Suppose there are n vertices then we will create one list which will keep information of all n vertices in the graph and after that we will create n lists, where each list will keep information of all adjacent vertices of that particular vertex. The structures of the nodes of these two lists would be-

```
struct Vertex
{
    int info;
    struct Vertex *nextVertex; /*next vertex in the linked list of vertices*/
    struct Edge *firstEdge; /*first Edge of the adjacency list of this vertex*/
}*start=NULL;
struct Edge
{
    struct Vertex *destVertex; /*Destination vertex of the Edge*/
    struct Edge *nextEdge; /*next Edge of the adjacency list*/
};
```

The following figure shows a directed graph and its adjacency list structure.

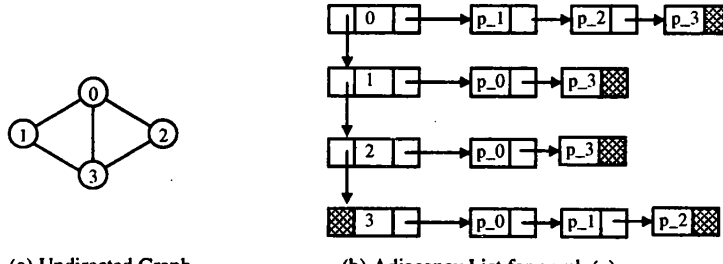


(a) Directed Graph (b) Adjacency List for graph (a)

Figure 7.25

In figure 7.25(b), on the left we have a linked list(vertically drawn) of the vertices of the graph. In the graph we have four vertices so there are four nodes in the linked list of vertices. For each vertex we have a separate list which stores pointers to adjacent vertices. For example the vertices 0, 2, 3 are adjacent to vertex 1 i.e. there are edges from vertex 1 to these vertices. So in the linked list of vertex 1, we have three nodes containing pointers to vertices 0, 2, 3. There is only one vertex adjacent to vertex 0, so in the list of vertex 0 there is only one node and it contains pointer to vertex 1. In the figure, p_0, p_1, p_2, p_3 represent pointers to nodes 0, 1, 2, 3 respectively.

We can have similar adjacency list for undirected graphs also. In the case of undirected graph, the space requirement doubles, since each edge appears in two lists. The following figure shows an undirected graph and its adjacency list structure.



(a) Undirected Graph (b) Adjacency List for graph (a)

Figure 7.26

7.10.2.1 Vertex insertion

Insertion of a vertex in an adjacency list only requires insertion of that vertex in the linked list of vertices. The new vertex is unconnected i.e. it has no edges. Edges to this new vertex have to be inserted separately. Let us insert a vertex 4 in the graph of figure 7.25.

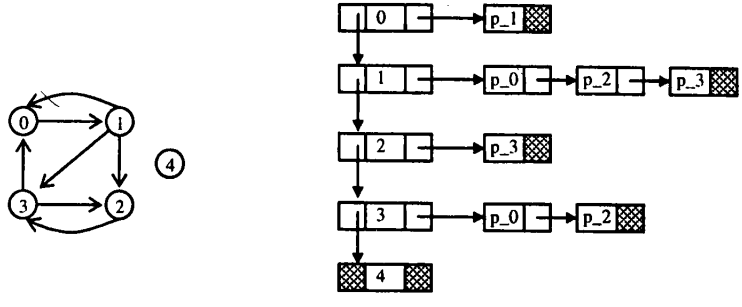


Figure 7.27

7.10.2.2 Edge insertion

Insertion of an edge requires insertion operation in the list of the starting vertex of edge. Suppose we want to add an edge (2, 0) in the graph of figure 7.25. For this we have to add a node in the edge list of vertex 2, and this new node will contain a pointer to vertex 0.

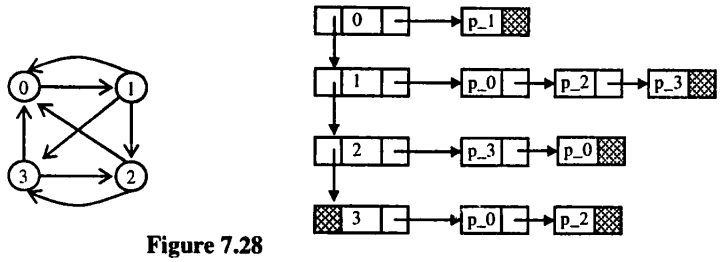


Figure 7.28

This was the procedure of edge insertion in directed graph. In undirected graph, the insertion operation has to be done in the lists of both start and end vertices of the edge.

7.10.2.3 Edge deletion

Deletion of an edge requires deletion operation in the list of the starting vertex of edge. Suppose we want to delete the edge (1,2) from graph of figure 7.25. For this, deletion will be performed in the list of vertex 1, and the node which will be deleted is the node which contains a pointer to vertex 2.

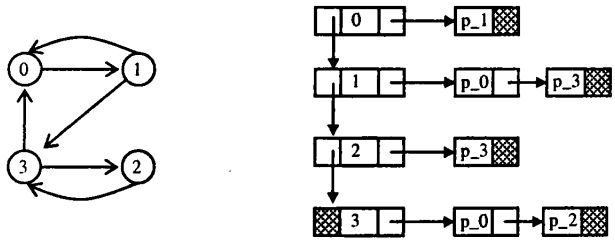


Figure 7.29

In an undirected graph the deletion operation has to be performed in lists of both start and end vertices of the edge.

7.10.2.4 Vertex deletion

Deletion of vertex requires deletion of that particular vertex from the linked list of vertices. Before deleting the node, it is necessary to delete all its incoming and outgoing edges.

Suppose we want to delete the vertex 2 from graph of figure 7.25, then first we will delete all edges where vertex 2 is the end node. For this we have to search the edge lists of all the vertices. The pointer to vertex 2 is found in adjacency lists of vertices 1 and 3. So it is deleted from there and hence the edges (1,2) and (3,2) are deleted from the graph. After this the adjacency list of vertex 2 is deleted which removes all edges where vertex 2 is the start node. In the graph of figure 7.25, there is only one node in adjacency list of vertex 2, so it is deleted and after that the vertex 2 is deleted from the linked list of vertices.

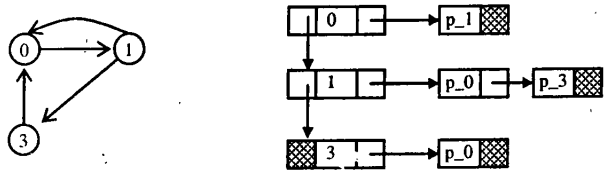


Figure 7.30

```
/*P7.3 Program for insertion and deletion of vertices and edges in a directed
```

```
graph using adjacency list*/
#include<stdio.h>
#include<stdlib.h>
struct Edge;
struct Vertex
{
    int info;
    struct Vertex *nextVertex; /*next vertex in the linked list of vertices*/
    struct Edge *firstEdge; /*first Edge of the adjacency list of this vertex*/
}*start = NULL;
struct Edge
{
     struct Vertex *destVertex;  /*Destination vertex of the Edge*/
     struct Edge *nextEdge; /*next Edge of the adjacency list*/
};
struct Vertex *findVertex(int u);
void insertVertex(int u);
void insertEdge(int u,int v);
void deleteEdge(int u,int v);
void deleteIncomingEdges(int u);
void deleteVertex(int u);
void display();
main()
{
       int choice,u,origin,destin;
       while(1)
       {
              printf("1.Insert a Vertex\n");
              printf("2.Insert an Edge\n");
              printf("3.Delete a Vertex\n");
              printf("4.Delete an Edge\n");
              printf("5.Display\n");
              printf("6.Exit\n");
              printf("Enter your choice : ");
              scanf("%d",&choice);
              switch(choice)
              {
               case 1:
                     printf("Enter a vertex to be inserted : ");
                     scanf("%d",&u);
                     insertVertex(u);
                     break;
               case 2:
                     printf("Enter an Edge to be inserted : ");
                     scanf("%d %d",&origin,&destin);
                     insertEdge(origin,destin);
                     break;
               case 3:
                     printf("Enter a vertex to be deleted : ");
                     scanf("%d",&u);
                     /*This function deletes all edges coming to this vertex*/
                     deleteIncomingEdges(u);
                     /*This function deletes the vertex from the vertex list*/
                     deleteVertex(u);
                     break;
               case 4:
                     printf("Enter an Edge to be deleted : ");
                     scanf("%d %d",&origin,&destin);
                     deleteEdge(origin,destin);
                     break;
               case 5:
                     display();
                     break;
               case 6:
```

```
                exit(1);
            default:
                printf("Wrong choice\n");
                break;
            }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
void insertVertex(int u)
{
        struct Vertex *tmp,*ptr;
        tmp = malloc(sizeof(struct Vertex));
        tmp->info = u;
        tmp->nextVertex = NULL;
        tmp->firstEdge = NULL;
        if(start==NULL)
        {
                start = tmp;
                return;
        }
        ptr = start;
        while(ptr->nextVertex!=NULL)
                ptr = ptr->nextVertex;
        ptr->nextVertex = tmp;
}/*End of insertVertex()*/
void deleteVertex(int u)
{
        struct Vertex *tmp,*q;
        struct Edge *p,*temporary;
        if(start==NULL)
        {
                printf("No vertices to be deleted\n");
                return;
        }
        if(start->info==u)/*Vertex to be deleted is first vertex of list*/
        {
                tmp = start;
                start = start->nextVertex;
        }
        else /*Vertex to be deleted is in between or at last*/
        {
                q = start;
                while(q->nextVertex!=NULL)
                {
                        if(q->nextVertex->info==u)
                                break;
                        q = q->nextVertex;
                }
                if(q->nextVertex==NULL)
                {
                        printf("Vertex not found\n");
                        return;
                }
                else
                {
                        tmp = q->nextVertex;
                        q->nextVertex = tmp->nextVertex;
                }
        }
        /*Before freeing the node tmp, free all edges going from this vertex*/
        p = tmp->firstEdge;
        while(p!=NULL)
        {
                temporary = p;
                p = p->nextEdge;
```

```
                free(temporary);
        }
        free(tmp);
}/*End of deleteVertex()*/
void deleteIncomingEdges(int u)
{
        struct Vertex *ptr;
        struct Edge *q,*tmp;
        ptr = start;
        while(ptr!=NULL)
        {
                if(ptr->firstEdge==NULL)    /*Edge list for vertex ptr is empty*/
                {
                        ptr = ptr->nextVertex;
                        continue; /*continue searching in other Edge lists*/
                }
                if(ptr->firstEdge->destVertex->info==u)
                {
                        tmp = ptr->firstEdge;
                        ptr->firstEdge = ptr->firstEdge->nextEdge;
                        free(tmp);
                        continue; /*continue searching in other Edge lists*/
                }
                q = ptr->firstEdge;
                while(q->nextEdge!= NULL)
                {
                        if(q->nextEdge->destVertex->info==u)
                        {
                                tmp = q->nextEdge;
                                q->nextEdge = tmp->nextEdge;
                                free(tmp);
                                continue;
                        }
                        q = q->nextEdge;
                }
                ptr = ptr->nextVertex;
        }/*End of while*/
}/*End of deleteIncomingEdges()*/
struct Vertex *findVertex(int u)
{
        struct Vertex *ptr,*loc;
        ptr = start;
        while(ptr!=NULL)
        {
                if(ptr->info==u)
                {
                        loc = ptr;
                        return loc;
                }
                else
                        ptr = ptr->nextVertex;
        }
        loc = NULL;
        return loc;
}/*End of findVertex()*/
void insertEdge(int u,int v)
{
        struct Vertex *locu,*locv;
        struct Edge *ptr,*tmp;
        locu = findVertex(u);
        locv = findVertex(v);
        if(locu==NULL)
        {
```

```
                printf("Start vertex not present, first insert vertex %d\n",u);
                return;
        }
        if(locv==NULL)
        {
                printf("End vertex not present, first insert vertex %d\n",v);
                return;
        }
        tmp = malloc(sizeof(struct Edge));
        tmp->destVertex = locv;
        tmp->nextEdge = NULL;
        if(locu->firstEdge==NULL)
        {
                locu->firstEdge = tmp;
               return;
        }
        ptr = locu->firstEdge;
        while(ptr->nextEdge!=NULL)
                ptr = ptr->nextEdge;
        ptr->nextEdge = tmp;
}/*End of insertEdge()*/
void deleteEdge(int u,int v)
{
        struct Vertex *locu;
        struct Edge *tmp,*q;
        locu = findVertex(u);
        if(locu==NULL)
        {
                printf("Start vertex not present\n");
                return;
        }
        if(locu->firstEdge==NULL)
        {
                printf("Edge not present\n");
                return;
        }
        if(locu->firstEdge->destVertex->info == v)
        {
                tmp = locu->firstEdge;
                locu->firstEdge = locu->firstEdge->nextEdge;
                free(tmp);
                return;
        }
        q = locu->firstEdge;
        while(q->nextEdge != NULL)
        {
                if(q->nextEdge->destVertex->info == v)
                {
                        tmp = q->nextEdge;
                        q->nextEdge = tmp->nextEdge;
                        free(tmp);
                        return;
                }
                q = q->nextEdge;
        }/*End of while*/
        printf("This Edge not present in the graph\n");
}/*End of deleteEdge()*/
void display()
{
        struct Vertex *ptr;
        struct Edge *q;
        ptr = start;
        while(ptr!=NULL)
        {
```

```
            printf("%d ->",ptr->info);
            q = ptr->firstEdge;
            while(q!=NULL)
            {
                    printf(" %d",q->destVertex->info);
                    q = q->nextEdge;
            }
            printf("\n");
            ptr = ptr->nextVertex;
        }
}/*End of display()*/
```

7.11 Transitive closure of a directed graph and Path Matrix

Transitive closure of a graph G is a graph G', where G' contains the same set of vertices as G and whenever there is a path from any vertex i to vertex j in G, there is an edge from i to j in G'.

The path matrix or reachability matrix of a graph G with n vertices is an n x n boolean matrix whose elements can be defined as-

$$P[i][j] = \begin{cases} 1 & \text{if there is a path from vertex i to vertex j} \\ 0 & \text{Otherwise.} \end{cases}$$

Thus the path matrix of a graph G is actually the adjacency matrix of its transitive closure G'. The path matrix of a graph is also known as the transitive closure matrix of the graph.

We will study two methods to compute the path matrix. The first method is by using powers of adjacency matrix and the second one is Warshall's algorithm. Here are some inferences that we can draw by looking at the path matrix-

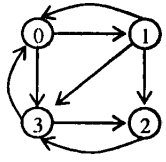
(i) If the element P[i][j] is equal to 1.
There is a path from vertex i to vertex j

(ii) If any main diagonal element i.e. any element P[i][i] in the path matrix is 1.
Graph contains a cycle

(iii) If all the elements in the path matrix are 1.
Graph is strongly connected

7:11.1 Computing Path matrix from powers of adjacency matrix

Let us take a graph and compute the path matrix for it from its adjacency matrix.



Adjacency Matrix A =

	0	1	2	3
0	0	1	0	1
1	1	0	1	1
2	0	0	0	1
3	1	0	1	0

Figure 7.31

Now we compute the matrix AM_2 by multiplying the adjacency matrix A with itself.

$AM_2 = A^2 =$

	0	1	2	3
0	2	0	2	1
1	1	1	1	2
2	1	0	1	0
3	0	1	0	2

In this matrix, value of $AM_2[i][j]$ will represent the number of paths of path length 2 from vertex i to vertex j. For example vertex 0 has two paths of path length 2 to vertex 2, and vertex 3 has two paths of path length 2 to itself, vertex 2 has one path of path length 2 to vertex 0, there is no path of path length 2 from vertex 2 to vertex

1. These paths may not be simple paths, i.e. all vertices in these paths need not be distinct. Now we compute the matrix AM_3 by multiplying the adjacency matrix A with AM_2.

$$AM_3 = AM_2 * A = A^3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 2 & 1 & 4 \\ 3 & 1 & 3 & 3 \\ 0 & 1 & 0 & 2 \\ 3 & 0 & 3 & 1 \end{bmatrix} \end{array}$$

Here $AM_3[i][j]$ will represent the number of paths of path length 3 from vertex i to vertex j. For example, vertex 0 has 4 paths of path length 3 to vertex 3(paths 0-3-2-3, 0-3-0-3, 0-1-2-3, 0-1-0-3) and vertex 3 has no path of path length 3 to vertex 1. Similarly, we can find out the matrix AM_4.

$$AM_4 = AM_3 * A = A^4 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 6 & 1 & 6 & 4 \\ 4 & 3 & 4 & 7 \\ 3 & 0 & 3 & 1 \\ 1 & 3 & 1 & 6 \end{bmatrix} \end{array}$$

In general we can say that if AM_k is equal to A^k, then any element $AM_k[i][j]$ represents the number of paths of path length k from vertex i to vertex j.

Let us define a matrix X where

$X = AM_1 + AM_2 + \ldots\ldots\ldots + AM_n$

X[i][j] denotes the number of paths, of path length n or less than n, from vertex i to vertex j. Here n is the total number of vertices in the graph.

For the graph in figure 7.31, the value of X will be-

$$X = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 9 & 4 & 9 & 10 \\ 9 & 5 & 9 & 13 \\ 4 & 1 & 4 & 4 \\ 5 & 4 & 5 & 9 \end{bmatrix} \end{array}$$

From definition of path matrix we know that P[i][j]=1 if there is a path from i to j, and this path can have length n or less than n. Now in the matrix X, if we replace all nonzero entries by 1 then we will get the path matrix or reachability matrix.

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 1 & 1 & 1 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

This graph is strongly connected since all the entries are equal to 1.

```
/*P7.4 Program to find out the path matrix by powers of adjacency matrix*/
#include<stdio.h>
#define MAX 100
void display(int matrix[MAX][MAX]);
void pow_matrix(int p,int adjp[MAX][MAX]);
void multiply(int mat1[MAX][MAX],int mat2[MAX][MAX],int mat3[MAX][MAX]);
void create_graph();
int adj[MAX][MAX];
int n;
main()
{
        int adjp[MAX][MAX];
        int x[MAX][MAX],path[MAX][MAX],i,j,p;
        create_graph();
```

```
        printf("The adjacency matrix is :\n");
        display(adj);
        /*Initialize all elements of matrix x to zero*/
        for(i=0; i<n; i++)
                for(j=0; j<n; j++)
                   x[i][j] = 0;
        /*All the powers of adj will be added to matrix x */
        for(p=1; p<=n; p++)
        {
                pow_matrix(p,adjp);
                printf("Adjacency matrix raised to power %d is - \n",p);
                display(adjp);
                for(i=0; i<n; i++)
                  for(j=0; j<n; j++)
                        x[i][j] = x[i][j]+adjp[i][j];
        }
        printf("The matrix x is :\n");
        display(x);
        for(i=0; i<n; i++)
                for(j=0; j<n; j++)
                        if(x[i][j]==0)
                                path[i][j] = 0;
                        else
                                path[i][j] = 1;

        printf("The path matrix is :\n");
        display(path);
}/*End of main()*/
void create_graph()
{
        int i,max_edges,origin,destin;
        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edges = n*(n-1);
        for(i=1; i<=max_edges; i++)
        {
                printf("Enter edge %d( -1 -1 ) to quit : ",i);
                scanf("%d %d",&origin,&destin);
                if((origin==-1) && (destin==-1))
                        break;
                if(origin>=n || destin>=n || origin<0 || destin<0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                        adj[origin][destin] = 1;
        }/*End of for*/
}/*End of create_graph()*/
/*This function computes the pth power of matrix adj and stores result in adjp*/
void pow_matrix(int p,int adjp[MAX][MAX])
{
        int i,j,k,tmp[MAX][MAX];
        /*Initially adjp is equal to adj*/
        for(i=0; i<n; i++)
          for(j=0; j<n; j++)
           adjp[i][j] = adj[i][j];
        for(k=1; k<p; k++)
        {
                /*Multiply adjp with adj and store result in tmp*/
                multiply(adjp,adj,tmp);
                for(i=0; i<n; i++)
                  for(j=0; j<n; j++)
                     adjp[i][j] = tmp[i][j]; /*New adjp is equal to tmp*/
```

```
        }
}/*End of pow_matrix()*/
/*This function multiplies mat1 and mat2 and stores the result in mat3*/
void multiply(int mat1[MAX][MAX],int mat2[MAX][MAX],int mat3[MAX][MAX])
{
        int i,j,k;
        for(i=0; i<n; i++)
                for(j=0; j<n; j++)
                {
                        mat3[i][j] = 0;
                        for(k=0; k<n; k++)
                                mat3[i][j] = mat3[i][j] + mat1[i][k] * mat2[k][j];
                }
}/*End of multiply()*/
void display(int matrix[MAX][MAX])
{
        int i,j;
        for(i=0; i<n; i++)
        {
                for(j=0; j<n; j++)
                        printf("%4d",matrix[i][j]);
                printf("\n");
        }
        printf("\n");
}/*End of display()*/
```

7.11.2 Warshall's Algorithm

We have seen that we can find the path matrix P of a given graph G by using powers of adjacency matrix. Warshall gave an efficient technique for finding path matrix of a graph known as Warshall's algorithm. Let us take a graph G of n vertices 0, 1, 2,.....n-1. We will define Boolean matrices P_{-1}, P_0, P_1, ... P_{n-1} where $P_k[i][j]$ is defined as-

$P_k[i][j] =$ { 1 If there is a simple path from vertex i to vertex j which does not use any intermediate vertex greater than k, i.e. all intermediate vertices belong to the set {0,1,.....k}

0 Otherwise

$P_{-1}[i][j] = 1$ If there is a simple path from vertex i to vertex j, which does not use any intermediate vertex.

$P_0[i][j] = 1$ If there is a simple path from vertex i to vertex j, which does not use any other intermediate vertex except possibly vertex 0.

$P_1[i][j] = 1$ If there is a simple path from vertex i to vertex j, which does not use any other intermediate vertex except possibly 0,1.

$P_2[i][j] = 1$ If there is a simple path from vertex i to vertex j which does not use any other intermediate vertices except possibly vertices 0,1,2.

....

....

$P_k[i][j] = 1$ If there is a simple path from vertex i to vertex j which does not use any other intermediate vertices except possibly 0,1,.....k

....

....

$P_{n-1}[i][j] = 1$ If there is a simple path from vertex i to vertex j which does not use any other intermediate vertices except possibly 0, 1, ...n-1.

Here P_{-1} represents the adjacency matrix and P_{n-1} represents the path matrix, let us see why.

$P_{-1}[i][j]=1$, If there is a simple path from vertex i to vertex j which does not use any vertex. The only way to go from i to j without using any vertex is to go directly from i to j. Hence $P_{-1}[i][j]=1$ means that there is an edge from i to j. So P_{-1} will be the adjacency matrix.

$P_{n-1}[i][j]=1$, If there is a simple path from i to j which does not use any vertices except 0, 1,...n-1. There are total n vertices means this path can use all n vertices; hence from the definition of path matrix we observe that P_{n-1} is the path matrix.

We know that P_{-1} is equal to the adjacency matrix, which we can easily find out from the graph. We have to find matrices $P_0, P_1, \ldots\ldots\ldots P_{n-1}$. If we know how to find matrix P_k from matrix P_{k-1} then we can find all these matrices. So now let us see how to find the value of $P_k[i][j]$ by looking at the matrix P_{k-1}.

If $P_{k-1}[i][j]$ is 1, $P_k[i][j]$ will also be 1, let us see why.

$P_{k-1}[i][j]=1$, implies that there is a simple path(say P1) from i to j which does not use any vertices except possibly 0, 1, ...k-1 or we can say that this path does not use any vertices numbered higher than k-1. So it is obvious that this path does not use any vertices numbered higher than k also or we can say that this path does not use any other vertices except possibly 0, 1, ... k. So $P_k[i][j]$ will be equal to 1.

Path P1 from i to j uses intermediate vertices from the set {0,1, ...k-1}
$P_{k-1}[i][j]=1$

Path P1 from i to j uses intermediate vertices from the set {0,1, ...k}
$P_k[i][j]=1$

Now we will consider the case when $P_{k-1}[i][j] = 0$. In this case $P_k[i][j]$ can be 0 or it can be 1. Let us find out the condition in which $P_k[i][j]$ will be 1 when $P_{k-1}[i][j]$ is 0.

$P_{k-1}[i][j]$ is 0, means there is no path from i to j using intermediate vertices 0,1, ...,k-1.
$P_k[i][j]$ is 1, means there is a path from i to j using intermediate vertices 0,1,.....k

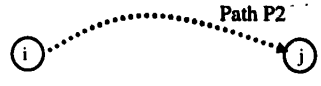
This means that when we use only vertices 0,1,.....,k-1 we have no path from i to j but when we use vertices 0,1,.....,k we get a path(say P2) from i to j. This path P2 will definitely pass through vertex k, so we can break it into two subpaths-

(i) path P2a from i to k using vertices 0,1,.....k-1.
(ii) path P2b from k to j using vertices 0,1,.....k-1.

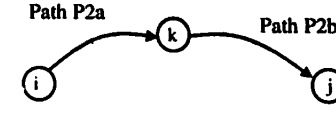
Since we have taken out k and the path P2 is simple(k can't be repeated), the paths P2a and P2b will have intermediate vertices from the set {0,1,2.....k-1}.

From path P2a we can write that $P_{k-1}[i][k]=1$
From path P2b we can write that $P_{k-1}[k][j]=1$

No path from i to j using intermediate vertices from the set {0,1,.....,k-1}
$P_{k-1}[i][j]=0$



Path P2 with intermediate vertices from the set {0,1,.....,k}
$P_k[i][j]=1$



Path P2 broken into two paths.
Paths P2a and paths P2b with intermediate vertices from the set {0,1,.....,k-1}
$P_{k-1}[i][k]=1$ and $P_{k-1}[k][j]=1$

For existence of path P2, the paths P2a and P2b should exist, i.e. for $P_k[i][j]$ to be 1 when $P_{k-1}[i][j]$ is zero, the values of $P_{k-1}[i][k]$ and $P_{k-1}[k][j]$ should be 1.

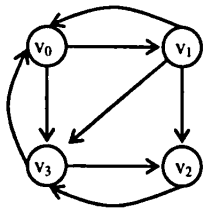
We can conclude that if $P_{k-1}[i][j]=0$ then $P_k[i][j]$ can be equal to 1 only
if $P_{k-1}[i][k]=1$ and $P_{k-1}[k][j]=1$.

So we have two situations when $P_k[i][j]$ can be 1

1. $P_{k-1}[i][j] = 1$ or
2. $P_{k-1}[i][k] = 1$ and $P_{k-1}[k][j] = 1$

To find any element $P_k[i][j]$ we will proceed as –
First see $P_{k-1}[i][j]$, If it is equal to 1, then $P_k[i][j]=1$, done
If $P_{k-1}[i][j]=0$, then see $P_{k-1}[i][k]$ and $P_{k-1}[k][j]$, if both are 1 then $P_k[i][j]=1$, done
Otherwise $P_k[i][j] = 0$

Let us take the same graph as in figure 7.31 and find out the values of P_{-1}, P_0, P_1, P_2, P_3



The first matrix P_{-1} is the adjacency matrix.

$$P_{-1} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Now we have to find the matrix P_0
To find any element $P_0[i][j]$ we will proceed as-
First see $P_{-1}[i][j]$, If it is equal to 1, then $P_0[i][j]=1$
Otherwise If $P_{-1}[i][j]=0$, then see $P_{-1}[i][0]$ and $P_{-1}[0][j]$, if both are 1 then $P_0[i][j]=1$
Otherwise $P_0[i][j]=0$

Calculation of some elements of matrix P_0-

* Find $P_0[2][3]$
  $P_{-1}[2][3] = 1$ so $P_0[2][3]=1$
* Find $P_0[3][1]$
  $P_{-1}[3][1] = 0$, so look at $P_{-1}[3][0]$ and $P_{-1}[0][1]$, both are 1 hence $P_0[3][1] = 1$
* Find $P_0[2][1]$
  $P_{-1}[2][1] = 0$, so look at $P_{-1}[2][0]$ and $P_{-1}[0][1]$, one of them is 0, hence $P_0[2][1] = 0$
* Find $P_0[2][2]$
  $P_{-1}[2][2] = 0$, so look at $P_{-1}[2][0]$ and $P_{-1}[0][2]$, both are 0, hence $P_0[2][2] = 0$

It is clear that if an entry is 1 in matrix P_{-1}, then it will also be 1 in P_0. So we can just copy all the 1's and see if the zero entries of P_{-1} can be changed to 1 in P_0. Changing of a zero entry to 1 implies that we get a path if we use 0 as the intermediate vertex.

$$P_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Now we have to find matrix P_1.

* Find $P_1[0][2]$
  $P_0[0][2] = 0$, so look at $P_0[0][1]$ and $P_0[1][2]$, both are 1, hence $P_1[0][2]=1$
* Find $P_1[2][1]$
  $P_0[2][1] = 0$, so look at $P_0[2][1]$ and $P_0[1][1]$, one of them is zero, hence $P_1[2][1]=0$

$$P_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

* Find $P_2[2][1]$

$P_1[2][1] = 0$, so look at $P_1[2][2]$ and $P_1[2][1]$, both are zero, hence $P_2[2][1]=0$

$$P_2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

And

$$P_3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Here P_{-1} is the adjacency matrix and P_3 is the path matrix of the graph. In the program all the calculation can be done in place using a single two dimensional array P.

```
/*P7.5 Program to find path matrix by Warshall's algorithm*/
#include<stdio.h>
#define MAX 100
void display(int matrix[MAX][MAX], int n);
int adj[MAX][MAX];
int n;
void create_graph();
main()
{
        int i,j,k;
        int P[MAX][MAX];
        create_graph();
        printf("The adjacency matrix is :\n");
        display(adj,n);
        for(i=0; i<n; i++)
           for(j=0; j<n; j++)
                 P[i][j] = adj[i][j];
        for(k=0; k<n; k++)
        {
                for(i=0; i<n; i++)
                  for(j=0; j<n; j++)
                      P[i][j] = (P[i][j] || (P[i][k] && P[k][j]));
                printf("P%d is :\n",k);
                display(P,n);
        }
        printf("P%d is the path matrix of the given graph\n",k-1);
}/*End of main() */
void display(int matrix[MAX][MAX],int n)
{
        int i,j;
        for(i=0; i<n; i++)
        {
                for(j=0; j<n; j++)
                        printf("%3d",matrix[i][j]);
                printf("\n");
        }
```

```
/*End of display()*/
void create_graph()
{
        int i,max_edges,origin,destin;
        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edges = n*(n-1);
        for(i=1; i<=max_edges; i++)
        {
                printf("Enter edge %d( -1 -1 ) to quit : ",i);
                scanf("%d %d",&origin,&destin);
                if((origin==-1) && (destin==-1))
                        break;
                if(origin>=n || destin>=n || origin<0 || destin<0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                        adj[origin][destin] = 1;
        }/*End of for*/
/*End of create_graph()*/
```

7.12 Traversal

Traversal in graph is different from traversal in tree or list because of the following reasons-

a) There is no first vertex or root vertex in a graph, hence the traversal can start from any vertex. We can choose any arbitrary vertex as the starting vertex. A traversal algorithm will produce different sequences for different starting vertices.

b) In tree or list, when we start traversing from the first vertex, all the elements are visited but in graph only those vertices will be visited which are reachable from the starting vertex. So if we want to visit all the vertices of the graph, we have to select another starting vertex from the remaining vertices in order to visit all the vertices left.

c) In tree or list while traversing, we never encounter a vertex more than once while in graph we may reach a vertex more than once. This is because in graph a vertex may have cycles and there may be more than one path to reach a vertex. So to ensure that each vertex is visited only once, we have to keep the status of each vertex whether it has been visited or not.

d) In tree or list we have unique traversals. For example if we are traversing a binary tree in inorder there can be only one sequence in which vertices are visited. But in graph, for the same technique of traversal there can be different sequences in which vertices can be visited. This is because there is no natural order among the successors of a vertex, and thus the successors may be visited in different orders producing different sequences. The order in which successors are visited may depend on the implementation.

Like binary trees, in graph also there can be many methods by which a graph can be traversed but two of them are standard and are known as breadth first search and depth first search.

7.12.1 Breadth First Search

In this technique, first we visit the starting vertex and then visit all the vertices adjacent to the starting vertex. After this we pick these adjacent vertices one by one and visit their adjacent vertices and this process goes on. This traversal is equivalent to level order traversal of trees.Let us take a graph and traverse it using breadth first traversal.

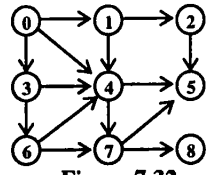
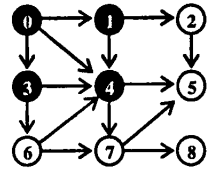


Figure 7.32

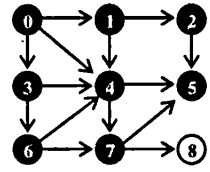
Let us take vertex 0 as the starting vertex. First we will visit the vertex 0. Then we will visit all vertices adjacent to vertex 0 i.e. 1, 4, 3. Here we can visit these three vertices in any order. Suppose we visit the vertices in order 1, 3, 4. Now the traversal is-

0 1 3 4



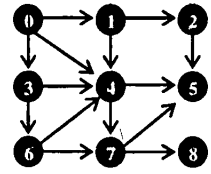
Now first we visit all the vertices adjacent to 1, then all the vertices adjacent to 3 and then all the vertices adjacent to 4. So first we will visit 2, then 6 and then 5, 7. Note that vertex 4 is adjacent to vertices 1 and 3, but it has already been visited so we've ignored it. Now the traversal is -

0 1 3 4 2 6 5 7



Now we will visit one by one all the vertices adjacent to vertices 2,6,5,7. We can see that vertex 5 is adjacent to vertex 2, but it has already been visited so we will just ignore it and proceed further. Now vertices adjacent to vertex 6 are vertices 4 and 7 which have already been visited so ignore them also. Vertex 5 has no adjacent vertices. Vertex 7 has vertices 5 and 8 adjacent to it out of which vertex 8 has not been visited, so visit vertex 8. Now the traversal is-

0 1 3 4 2 6 5 7 8



Now we have to visit vertices adjacent to vertex 8 but there is no vertex adjacent to vertex 8 so our procedure stops.

This was the traversal when we take vertex 0 as the starting vertex. Suppose we take vertex 1 as the starting vertex. Then applying above technique, we will get the following traversal-

1 2 4 5 7 8

Here are different traversals when we take different starting vertices.

Start Vertex	Traversal
0	0 1 3 4 2 6 5 7 8
1	1 2 4 5 7 8
2	2 5
3	3 4 6 5 7 8
4	4 5 7 8
5	5
6	6 4 7 5 8
7	7 5 8
8	8

Note that these traversals are not unique, there can be different traversals depending on the order in which we visit the successors.

We can see that all the vertices are not visited in some cases. The vertices which are visited are those vertices which are reachable from starting vertex. So to make sure that all the vertices are visited we need to repeat the same procedure for each unvisited vertex in the graph. Breadth first search is implemented through queue.

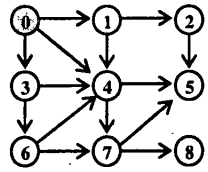
7.12.1.1 Implementation of Breadth First Search using queue

During the algorithm, any vertex will be in one of the three states - initial, waiting, visited. At the start of the algorithm all vertices will be in initial state, when a vertex will be inserted in the queue its state will change from initial to waiting. When a vertex will be deleted from queue and visited, its state will change from waiting to visited. The procedure is as-

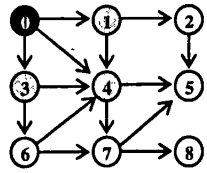
Initially queue is empty, and all vertices are in initial state.
1. Insert the starting vertex into the queue, change its state to waiting.
2. Delete front element from the queue and visit it, change its state to visited.
3. Look for the adjacent vertices of the deleted element, and from these insert only those vertices into the queue which are in the initial state. Change the state of all these inserted vertices from initial to waiting.
4. Repeat steps 2, 3 until the queue is empty.

Let us take vertex 0 as the starting vertex for traversal in the graph of figure 7.32. In each step we will show the traversal and the contents of queue. In the figure, the different states of the vertices are shown by different colors. White color indicates initial state, grey indicates waiting state and black indicates visited state.

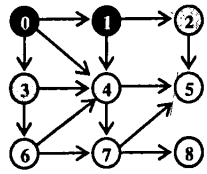
(i) Insert the vertex 0 into the queue.
Queue : 0



(ii) Delete vertex 0 from queue, and visit it.
Traversal : 0
Vertices adjacent to vertex 0 are vertices 1, 3, 4 and all of these are in initial state, so insert them into the queue.
Queue : 1, 3, 4

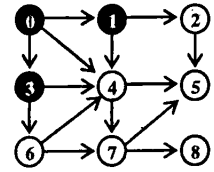


(iii) Delete vertex 1 from queue, and visit it.
Traversal : 0, 1
Vertices adjacent to vertex 1 are vertices 2 and 4. Vertex 4 is in waiting state because it is in the queue, so it is not inserted in the queue. Vertex 2 is in initial state, so insert it into the queue.
Queue : 3, 4, 2

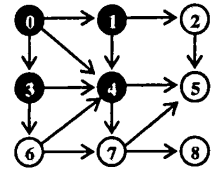


Here we can see why we have taken the concept of waiting state. Vertex 4 is in waiting state i.e. it is already present in the queue so it is not inserted into the queue. The concept of waiting state helps us avoid insertion of duplicate vertices in the queue.

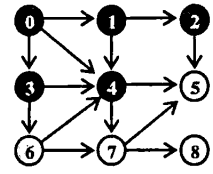
(iv) Delete vertex 3 from queue, and visit it.
Traversal : 0, 1, 3
Vertices adjacent to vertex 3 are vertices 4 and 6.
Vertex 4 is in the waiting state and vertex 6 is in initial state, so insert only vertex 6 into the queue.
Queue : 4, 2, 6



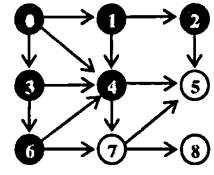
(v) Delete vertex 4 from the queue, and visit it.
Traversal : 0, 1, 3, 4
Vertices adjacent to vertex 4 are vertices 5 and 7, and both are in initial state so insert them into the queue.
Queue : 2, 6, 5, 7



(vi) Delete vertex 2 from the queue, and visit it.
Traversal : 0, 1, 3, 4, 2
Vertex adjacent to vertex 2 is vertex 5.
Vertex 5 is in waiting state because it is already present in the queue, so it is not inserted into the queue.
Queue : 6, 5, 7

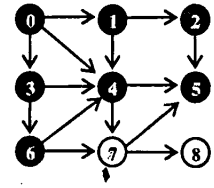


(vii) Delete vertex 6 from the queue, and visit it.
Traversal : 0, 1, 3, 4, 2, 6,
Vertices adjacent to vertex 6 are vertices 4 and 7.
Vertex 4 is in visited state and vertex 7 is in waiting state so nothing is inserted into the queue.
Queue : 5, 7

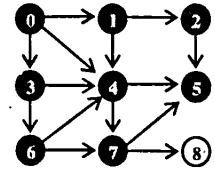


Here we can see why we have taken the concept of visited state. Vertex 4 is in visited state i.e. it has been included in the traversal, so there is no need of its insertion into the queue. The concept of visited state helps us avoid visiting a vertex more than once.

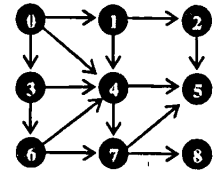
(viii) Delete vertex 5 from queue, and visit it.
Traversal : 0, 1, 3, 4, 2, 6, 5
Vertex 5 has no adjacent vertices.
Queue : 7



(ix) Delete 7 from queue, and visit it.
Traversal : 0, 1, 3, 4, 2, 6, 5, 7
Vertices adjacent to vertex 7 are vertices 5 and 8.
Vertex 5 is in visited state and vertex 8 is in initial state, so insert only vertex 8 into the queue.
Queue : 8



(x) Delete vertex 8 from queue, and visit it.
Traversal : 0, 1, 3, 4, 2, 6, 5, 7, 8
There is no vertex adjacent to vertex 8
Queue : EMPTY



Now the queue is empty so we will stop our process. This way we get the breadth first traversal when vertex 0 is taken as the starting vertex.

```
/*P7.6 Program for traversing a directed graph through BFS, visiting only those vertices that
are reachable from start vertex*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define initial 1
#define waiting 2
#define visited 3
int n;     /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Adjacency Matrix*/
int state[MAX]; /*can be initial, waiting or visited*/
void create_graph();
void BF_Traversal();
void BFS(int v);
int queue[MAX],front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();
main()
{
        create_graph();
        BF_Traversal();
}/*End of main()*/
void BF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
                state[v]=initial;
        printf("Enter starting vertex for Breadth First Search : ");
        scanf("%d",&v);
        BFS(v);
}/*End of BF_Traversal()*/
void BFS(int v)
{
        int i;
        insert_queue(v);
        state[v]=waiting;
        while(!isEmpty_queue())
        {
                v = delete_queue();
                printf("%d ",v);
                state[v]=visited;
                for(i=0; i<n; i++)
                {
                        /*Check for adjacent unvisited vertices*/
                        if(adj[v][i]==1 && state[i]==initial)
                        {
                                insert_queue(i);
                                state[i] = waiting;
                        }
                }
        }
        printf("\n");
}/*End of BFS()*/
void insert_queue(int vertex)
{
        if(rear==MAX-1)
                printf("Queue Overflow\n");
        else
```

```
        {
                if(front==-1)  /*If queue is initially empty*/
                        front = 0;
                rear = rear+1;
                queue[rear] = vertex ;
        }
}/*End of insert_queue()*/
int isEmpty_queue()
{
        if(front==-1 || front>rear)
                return 1;
        else
                return 0;
}/*End of isEmpty_queue()*/
int delete_queue()
{
        int del_item;
        if(front==-1 || front>rear)
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        del_item = queue[front];
        front = front+1;
        return del_item;
}/*End of delete_queue()*/
```

The function `create_graph()` is same as in program P7.2.

This process can visit only those vertices which are reachable from the starting vertex. For example if we start traversing from vertex 4 instead of vertex 0, then all the vertices will not be visited. The traversal would be – 4 5 7 8.

If we want to visit all the vertices, then we can take any unvisited vertex as starting vertex and again start breadth first search from there. This process will continue until all the vertices are visited. So if want to visit all the vertices when 4 is the start vertex, then we have to select another start vertex after visiting 5, 7 and 8. Suppose we take 0 as the next start vertex, so now the traversal would be – 4 5 7 8 0 1 3 2 6. Now all the vertices are visited so there is no need to choose any other start vertex.

In the program, we have to make a small addition in the `BF_Traversal()` function. After the call to `BFS()` with start vertex, we will check all vertices one by one in a loop, and if we get any vertex that is in initial state we will call `BFS()` with that vertex.

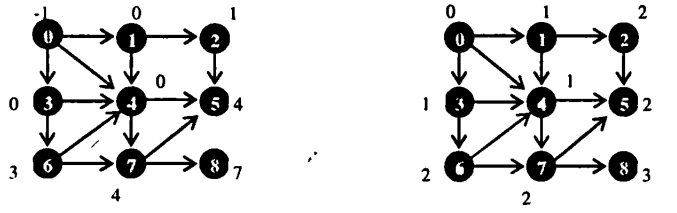
```
void BF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
                state[v]=initial;
        printf("Enter starting vertex for Breadth First Search : ");
        scanf("%d", &v);
        BFS(v);
        for(v=0; v<n; v++)
                if(state[v]==initial)
                        BFS(v);
}/*End of BF_Traversal()*/
```

Now suppose that while performing breadth first search, we assign two values to each vertex in the graph – a predecessor and a distance value. Whenever we insert a vertex in the queue we set its predecessor and distance values.

The predecessor of starting vertex is taken as NIL(-1). The distance value of starting vertex is taken as 0 and the distance value of any other vertex is one more than the distance value of its predecessor. If we take the case of example described in section 7.12.1.1, then the predecessor and distance values set in different steps would be-

In step (i) pred[0] = -1
d[0] = 0
In step (ii) pred[1] = pred[3] = pred[4] = 0
d[1] = d[3] = d[4] = d[0] + 1 = 0 + 1 = 1
In step (iii) pred[2] = 1
d[2] = d[1]+1 = 1+1 = 2
In step (iv) pred[6]=3
d[6] = d[3]+1 = 1+1 = 2
In step (v) pred[5] = pred[7] = 4
d[5] = d[7] = d[4]+1 = 1+1 =2
In step (ix) pred[8] = 7
d[8] = d[7] +1 = 2+1 = 3

In the steps (vi), (vii), (viii), (x) nothing is inserted in the queue so no value is set in these steps. The following figures show the graph with predecessor and distance values.



(a) Graph with predecessor values (b) Graph with distance values

Figure 7.33

The distance value of a vertex u gives us the shortest distance (number of edges) of u from the starting vertex. For example the shortest distance from vertex 0 to vertex 8 is 3. This shortest path can be obtained by following the predecessors values till we get the start vertex as a predecessor. Let us see how we can get this path in the case of 8. Predecessor of 8 is 7, predecessor of 7 is 4, predecessor of 4 is 0, so the shortest path is 0 4 7 8.

There may be other paths from vertex 0 to vertex 8 but length of none of them would be less than 3. Hence breadth first search can be used to find the shortest distances to all vertices reachable from the start vertex in unweighted graphs. The length of shortest path is given by distance value and this shortest path can be obtained by following the predecessor values.

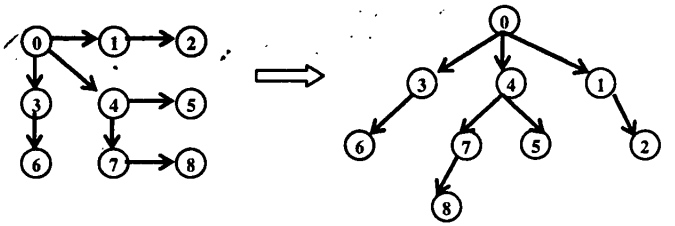
```
/*P7.6b Program for traversing a directed graph through BFS,and finding shortest distance and
shortest path of any vertex from start vertex*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define infinity 9999
#define NIL -1
#define initial 1
#define waiting 2
#define visited 3
int n;    /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Adjacency Matrix*/
int state[MAX]; /*can be initial, waiting or visited*/
int distance[MAX];
int predecessor[MAX];
void create_graph();
void BF_Traversal();
void BFS(int v);
int queue[MAX],front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();
main()
```

```
{
        int u,v,i,count,path[MAX];
        create_graph();
        BF_Traversal();
        while(1)
        {
                printf("Enter destination vertex(-1 to quit) : ");
                scanf("%d",&v);
                if(v<-1 || v>n-1)
                {
                        printf("Destination vertex does not exist\n");
                        continue;
                }
                if(v==-1)
                        break;
                count = 0;
                if(distance[v]==infinity)
                {
                  printf("No path from start vertex to destination vertex\n");
                  continue;
                }
                else
                   printf("Shortest distance is : %d\n", distance[v]);
                /*Store the full path in array path*/
                while(v!=NIL)
                {
                        count++;
                        path[count] = v;
                        u = predecessor[v];
                        v = u;
                }
                printf("Shortest Path is : ");
                for(i=count; i>1; i--)
                        printf("%d->",path[i]);
                printf("%d",path[i]);
                printf("\n");
        }
}/*End of main()*/
void BF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
        {
                state[v] = initial;
                predecessor[v] = NIL;
                distance[v] = infinity;
        }
        printf("Enter starting vertex for Breadth First Search : ");
        scanf("%d",&v);
        BFS(v);
        printf("\n");
}/*End of BF_Traversal()*/
void BFS(int v)
{
        int i;
        insert_queue(v);
        state[v]=waiting;
        distance[v]=0;
        predecessor[v]=NIL;
        while(!isEmpty_queue())
        {
                v = delete_queue();
                state[v] = visited;
                for(i=0; i<n; i++)
```

```
            {
                    /*Check for adjacent unvisited vertices*/
                    if(adj[v][i]==1 && state[i]==initial)
                    {
                            insert_queue(i);
                            state[i]=waiting;
                            predecessor[i] = v;
                            distance[i] = distance[v]+1;
                    }
            }
        }
}/*End of BFS()*/
```

In the program, we initialize the distance values of all vertices to infinity (a very large number), and the predecessor values are initialized to NIL(-1). Since our vertices start from 0, we can take the value of NIL equal to -1. In the function `BF_traversal()`, we don't write the for loop as we don't have to find out shortest distances and paths to vertices that are not reachable.

Now let's look at the graph with predecessor values. If we draw only those edges which join a vertex to its predecessor then we get the predecessor subgraph which is a spanning tree of the graph. This spanning tree is called the breadth first search spanning tree.

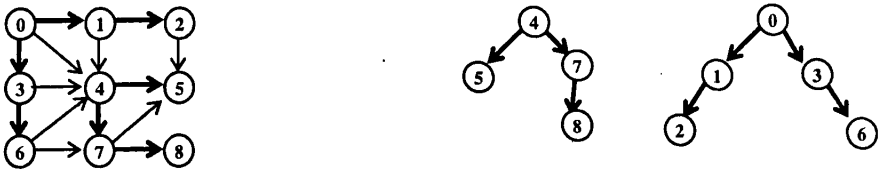


(a) (b)

Figure 7.34

The figure 7.34(a) shows the breadth first spanning tree. This figure is redrawn on the right so that it looks like a tree with vertex 0 as the root.

If all vertices are not reachable form start vertex then we get a BFS spanning forest having more than one spanning tree. For example if in the above graph we start traversing from vertex 4 then we get two spanning trees. Starting at vertex 4 we can visit only vertices 5,7,8 as these are the only vertices reachable from 4. After this we select vertex 0 as the next start vertex and then rest of the vertices are visited. So we get a BFS spanning forest consisting of two spanning trees.



(a) Graph traversed with start vertices 4 and 0 (b) Spanning Forest

Figure 7.35

In the spanning forest, the arbitrarily chosen vertex is the root of the spanning tree. All the edges that are in the spanning forest are called tree edges. We can output all the tree edges by adding a small line in the BFS function-

```
void BFS(int v)
{
        int i;
        insert_queue(v);
        state[v] = waiting;
        while(!isEmpty_queue())
        {
                v = delete_queue();
```

```
                printf("Vertex %d visited\n",v);
                state[v] = visited;
                for(i=0; i<n; i++)
                {
                        if(adj[v][i]==1 && state[i]==initial)
                        {
                                insert_queue(i);
                                state[i] = waiting;
                                printf("-----Tree edge - (%d,%d)\n",v,i);
                        }
                }
        }
}/*End of BFS()*/
```

Breadth first search in an undirected graph is performed in the same manner as in a directed graph. Consider the undirected graph given below.

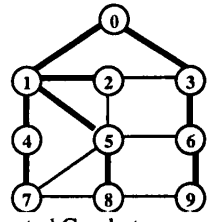
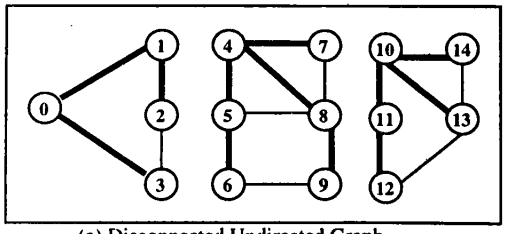


Figure 7.36 Undirected Graph, traversed with start vertex 0

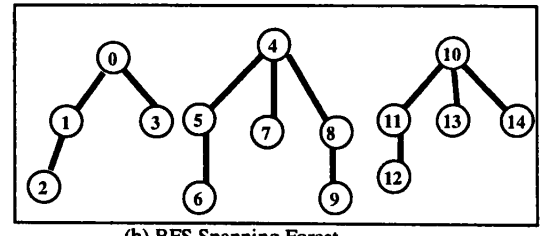
If we take 0 as start vertex, the breadth first traversal would be-
0 1 3 2 4 5 6 7 8 9

As stated before, this traversal is not unique; there may be other traversals depending on the order of visiting of successors.

If the undirected graph is connected, then we can reach all the vertices taking any vertex as the start vertex. If the graph is not connected, then we can visit only those vertices which are in the same connected component as the start vertex. So we can pick another unvisited vertex and start traversing from there and continue this procedure till all vertices are visited. The following figure shows a disconnected undirected graph and its BFS spanning forest. There is a spanning tree corresponding to each connected component of the graph.



(a) Disconnected Undirected Graph



(b) BFS Spanning Forest

Figure 7.37

The breadth first traversal of the graph of figure 7.37(a) taking 0, 4 and 10 as the start vertices is -
0 1 3 2 4 5 7 8 6 9 10 11 13 14 12

An application of BFS could be to find out whether an undirected graph is connected or not. An undirected graph is connected if we can visit all the vertices taking any arbitrary start vertex, and there is no need to choose another start vertex.

In the program, after calling function `BFS()` for the start vertex, if even a single vertex is left in initial state then the graph is not connected.

```
void BF_Traversal()
{
```

```
        int v;
        int connected = 1;
        for(v=0; v<n; v++)
                state[v]=initial;
        BFS(0); /*start BFS from vertex 0*/
        for(v=0; v<n; v++)
        {
               if(state[v]==initial)
               {
                       connected = 0;
                       break;
               }
        }
        if(connected)
        printf("Graph is connected\n");
        else
                printf("Graph is not connected\n");
}/*End of BF_Traversal()*/
```

We can find all the connected components using breadth first search. To find the connected components, all vertices in the graph are given a label such that vertices in same component get the same label. For example in the graph 7.37(a) we have three connected components, all vertices in first component have label 1, vertices in second component have label 2, vertices in third component have label 3.

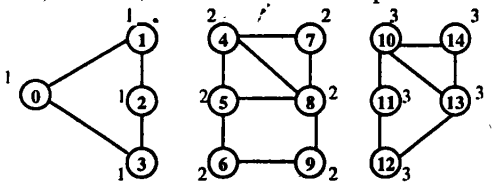


Figure 7.38

```
/*P7.6f Program to find connected components in an undirected graph*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define initial 1
#define waiting 2
#define visited 3
int n;     /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Adjacency Matrix*/
int state[MAX]; /*can be initial, waiting or visited*/
int label[MAX]; /*Denotes the Component Number*/
void create_graph();
void BF_Traversal();
void BFS(int v, int component_Num);
int queue[MAX],front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();
main()
{
        create_graph();
        BF_Traversal();
}/*End of main()*/
void BF_Traversal()
{
        int v,components = 0;
        for(v=0;v<n;v++)
                state[v] = initial;
        components++;
```

```
        BFS(0,components); /*start BFS from vertex 0*/
        for(v=0; v<n; v++)
        {
                if(state[v]==initial)
                {
                        components++;
                        BFS(v,components);
                }
        }
        printf("Number of connected components = %d\n", components);
        if(components==1)
                printf("Connected Graph\n");
        else
                printf("Not a Connected Graph\n");
}/*End of BF_Traversal()*/
void BFS(int v,int component_Num)
{
        int i;
        insert_queue(v);
        state[v] = waiting;
        while(!isEmpty_queue())
        {
                v = delete_queue();
                state[v] = visited;
                label[v] = component_Num;
                printf("Vertex %d  Component = %d\n",v,label[v]);
                for(i=0; i<n; i++)
                {
                        /*Check for adjacent unvisited vertices*/
                        if(adj[v][i]==1 && state[i]==initial)
                        {
                                insert_queue(i);
                                state[i] = waiting;
                        }
                }
        }
        printf("\n");
}/*End of BFS()*/
```

7.12.2 Depth First Search

Traversal using depth first search is like traveling a maze. We travel along a path in the graph and when a dead end comes we backtrack. This technique is named so because search proceeds deeper in the graph i.e. we traverse along a path as deep as we can.

First the starting vertex is visited and then we will pick up any path that starts from the starting vertex and visit all the vertices in this path till we reach a dead end. This means visit the starting vertex(say v1) and then any vertex adjacent to it(say v2). Now if v2 has any vertex adjacent to it which has not been visited then visit it, and so on till we come to a dead end. Dead end means that we reach a vertex which does not have any adjacent vertex or all of its adjacent vertices have been visited. After reaching the dead end we will backtrack along the path that we have visited till now. Suppose the path that we've traversed is v1-v2-v3-v4-v5. After traversing v5 we reached a dead end. Now we will move backwards till we reach a vertex that has any unvisited adjacent vertex. We move back and reach v4 but see that it has no unvisited adjacent vertices so we will reach vertex v3. Now if v3 has an unvisited vertex adjacent to it, we will pick up a path that starts from v3 and visit it until we reach a dead end. Then again we will backtrack. This procedure finishes when we reach the starting vertex and there are no vertices adjacent to it which have to be visited.

Let us take a graph and perform a depth first search taking vertex 0 as the start vertex. The successors of a vertex can be visited in any order.

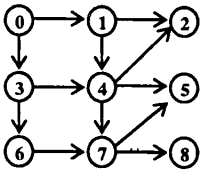
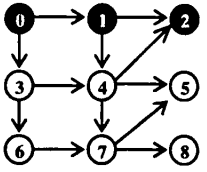


Figure 7.39

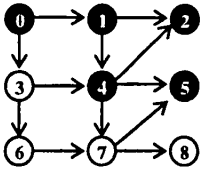
First, we will visit vertex 0. Vertices adjacent to vertex 0 are 1 and 3. Suppose we visit vertex 1. Now we look at the adjacent vertices of 1; from the two adjacent vertices 2 and 4 we choose to visit 2. Till now the traversal is -

0 1 2



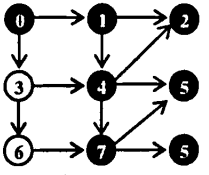
There is no vertex adjacent to vertex 2, means we have reached the end of the path or a dead end from where we can't go forward. So we will move backward. We reach vertex 1 and see if there is any vertex adjacent to it, and not visited yet. Vertex 4 is such a vertex and therefore we visit it. Now vertices 5 and 7 are adjacent to 4 and unvisited, and from these we choose to visit vertex 5. Till now the traversal is -

0 1 2 4 5



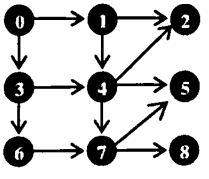
There is no vertex adjacent to vertex 5 so we will backtrack. We reach vertex 4, and its unvisited adjacent vertex is 7 so we visit it. Now vertex 8 is the only unvisited vertex adjacent to 7 so we visit it. Till now the traversal is -

0 1 2 4 5 7 8



Vertex 8 has no unvisited adjacent vertex so we backtrack and reach vertex 7. Now vertex 7 also has no unvisited adjacent vertex so we backtrack and reach vertex 4. Vertex 4 also has no unvisited adjacent vertex so we backtrack and reach vertex 1. Vertex 1 also has no unvisited adjacent vertex so we backtrack and reach vertex 0. Vertex 3 is adjacent to vertex 0 and is unvisited so we visit vertex 3. Vertex 6 is adjacent to vertex 3 and is unvisited so we visit vertex 6. Till now the traversal is -

0 1 2 4 5 7 8 3 6



Now vertex 6 has no unvisited adjacent vertex so we backtrack and reach vertex 3. Vertex 3 also has no unvisited adjacent vertex so we backtrack and reach vertex 0. Vertex 0 also has no unvisited adjacent vertex left and it is the start vertex so now we can't backtrack and hence our traversal finishes. The traversal is-
0 1 2 4 5 7 8 3 6.

Depth first search can be implemented through stack or recursively.

7.12.2.1 Implementation of Depth First Search using stack

During the algorithm any vertex will be in one of the two states – initial or visited. At the start of the algorithm, all vertices will be in initial state, and when a vertex will be popped from stack its state will change to visited. The procedure is as-

Initially stack is empty, and all vertices are in initial state.
1. Push starting vertex on the stack.
2. Pop a vertex from the stack.
3. If popped vertex is in initial state, visit it and change its state to visited. Push all unvisited vertices adjacent to the popped vertex.
4. Repeat steps 2 and 3 until stack is empty.

There is no restriction on the order in which the successors of a vertex are visited and so we can push the successors of a vertex in any order. Here we are pushing the successors in descending order of their numbers. For example if the successors are 2, 4, 6 then we will push 6 first and then 4 and then 2. Let us find the depth first traversal of the following graph using stack.

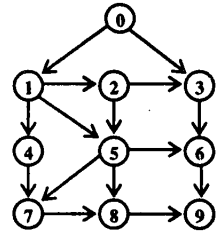


Figure 7.40

Start vertex is 0, so initially push vertex 0 on the stack.

Pop 0:	Visit 0	Push 3, 1	Stack : 3 1
Pop 1:	Visit 1	Push 5, 4, 2	Stack : 3 5 4 2
Pop 2:	Visit 2	Push 5, 3	Stack : 3 5 4 5 3
Pop 3:	Visit 3	Push 6	Stack : 3 5 4 5 6
Pop 6:	Visit 6	Push 9	Stack : 3 5 4 5 9
Pop 9:	Visit 9		Stack : 3 5 4 5
Pop 5:	Visit 5	Push 8, 7	Stack : 3 5 4 8 7
Pop 7:	Visit 7	Push 8	Stack : 3 5 4 8 8
Pop 8:	Visit 8		Stack : 3 5 4 8
Pop 8:			Stack : 3 5 4
Pop 4:	Visit 4		Stack : 3 5
Pop 5:			Stack : 3
Pop 3:			Stack : Empty

Depth first traversal is : 0 1 2 3 6 9 5 7 8 4

In BFS, if a vertex was already present in the queue then it was not inserted again in the queue. So there we changed the state of vertex from initial to waiting as soon as it was inserted in the queue, and never inserted a

vertex in the queue that was in waiting state. In DFS, we don't have the concept of waiting state and so there may be multiple copies of a vertex in the stack.

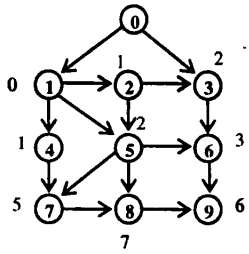
If we don't insert a vertex already present in the stack, then we will not be able to visit the vertices in depth first search order. For example if we traverse the graph of figure 7.40 in this manner then we get the traversal as 0 1 2 4 7 8 9 5 6 3, which is clearly not in depth first order.

```
/*P7.7 Program for traversing a directed graph through DFS, visiting only vertices reachable
from start vertex*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define initial 1
#define visited 2
int n;    /* Number of nodes in the graph */
int adj[MAX][MAX]; /*Adjacency Matrix*/
int state[MAX]; /*Can be initial or visited */

void DF_Traversal();
void DFS(int v);
void create_graph();
int stack[MAX];
int top = -1;
void push(int v);
int pop();
int isEmpty_stack();

main()
{
        create_graph();
        DF_Traversal();
}/*End of main()*/

void DF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
                state[v]=initial;
        printf("Enter starting node for Depth First Search : ");
        scanf("%d",&v);
        DFS(v);
}/*End of DF_Traversal( )*/

void DFS(int v)
{
        int i;
        push(v);
        while(!isEmpty_stack())
        {
                v = pop();
                if(state[v]==initial)
                {
                        printf("%d ",v);
                        state[v]=visited;
                }
                for(i=n-1; i>=0; i--)
                {
                        if(adj[v][i]==1 && state[i]==initial)
                                push(i);
                }
        }
}/*End of DFS( )*/

void push(int v)
{
        if(top==(MAX-1))
        {
```

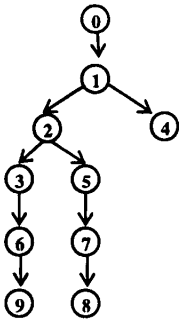
```
                printf("Stack Overflow\n");
                return;
        }
        top=top+1;
        stack[top] = v;
}/*End of push()*/
int pop()
{
        int v;
        if(top==-1)
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        else
        {
                v = stack[top];
                top=top-1;
                return v;
        }
}/*End of pop()*/
int isEmpty_stack( )
{
  if(top==-1)
          return 1;
  else
          return 0;
}/*End if isEmpty_stack()*/
void create_graph()
{
        int i,max_edges,origin,destin;
        printf("Enter number of nodes : ");
        scanf("%d",&n);
        max_edges = n*(n-1);

        for(i=1; i<=max_edges; i++)
        {
                printf("Enter edge %d( -1 -1 to quit ) : ",i);
                scanf("%d %d",&origin,&destin);
                if((origin==-1) && (destin==-1))
                        break;
                if(origin>=n || destin>=n || origin<0 || destin<0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                        adj[origin][destin] = 1;
        }
}/*End of create_graph()*/
```

If all vertices are not reachable from the start vertex then we need to repeat the procedure taking some other start vertex. This is similar to the process we had done in breadth first search. In the function `DF_Traversal()`, we will add a loop which will check the state of all vertices after the first DFS.

As in BFS, here also we can assign a predecessor to each vertex and get the predecessor subgraph which would be a spanning tree or spanning forest of the given graph depending on the reachability of all vertices from the start vertex.



Graph with Predecessor of each vertex



Spanning Tree

Figure 7.41

```
void DF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
        {
                state[v] = initial;
                predecessor[v] = NIL;
        }
        printf("Enter starting vertex for Depth First Search : ");
        scanf("%d",&v);
        DFS(v);
        for(v=0; v<n; v++)
        {
                if(state[v]==initial)
                        DFS(v);
        }
        printf("\n");
        for(v=0; v<n; v++)
          printf("Vertex %d, Predecessor %d\n", v,predecessor[v]);
        printf("Tree Edges : ");
        for(v=0; v<n; v++)
        {
                if(predecessor[v]!=-1)
                    printf("Tree edge : %d->%d\n", predecessor[v], v);
        }
}/*End of DF_Traversal( )*/
void DFS(int v)
{
        int i;
        push(v);
        while(!isEmpty_stack())
        {
                v = pop();
                if(state[v]==initial)
                {
                   printf("%d ",v);
                   state[v] = visited;
                }
                for(i=n-1; i>=0; i--)
                {
                        if(adj[v][i]==1 && state[i]==initial)
                        {
                                push(i);
                                predecessor[i] = v;
                        }
                }
```

```
        }
}/*End of DFS( )*/
```

For an undirected graph, the depth first search will proceed in the same manner. Like breadth first search, we can use depth first search also to find whether a graph is connected or not and for finding all the connected components.

7.12.2.2 Recursive Implementation of Depth First Search

In the recursive implementation we will take the three states of a vertex as initial, visited and finished. At the start all vertices are in initial state, after a vertex is visited its state becomes visited. The state of a vertex becomes finished when we backtrack from it i.e. if it has been visited and it has no adjacent vertices or all its adjacent vertices are in finished state.

```
/*P7.8 Program for traversing a directed graph through DFS recursively*/
#include<stdio.h>
#define MAX 100
#define initial 1
#define visited 2
#define finished 3
int n;
int adj[MAX][MAX];
void create_graph();
int state[MAX];
void DF_Traversal();
void DFS(int v);

main()
{
        create_graph();
        DF_Traversal();
}/*End of main()*/

void DF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
                state[v] = initial;
        printf("Enter starting vertex for Depth First Search : ");
        scanf("%d",&v);
        DFS(v);
        for(v=0; v<n; v++)
        {
                if(state[v]==initial)
                        DFS(v);
        }
        printf("\n");
}/*End of DF_Traversal()*/

void DFS(int v)
{
        int i;
        printf("%d ",v);
        state[v] = visited;
        for(i=0; i<n; i++)
        {
                if(adj[v][i]==1 && state[i]==initial)
                        DFS(i);
        }
        state[v] = finished;
}/*End of DFS()*/
```

The figure 7.42 shows the depth first search for the graph of figure 7.39, vertices which are visited are in grey color and vertices which are finished are in black color. We have also given two numbers to each vertex, discovery time and finishing time. The discovery time of a vertex is assigned when the vertex is first discovered and visited i.e. when it becomes grey. The finishing time of a vertex is assigned when we backtrack from it, and it becomes black. As earlier, we have chosen to visit the successors in ascending order of their number i.e. if a vertex has 1 and 3 as successors then first we will visit 1 and then 3.

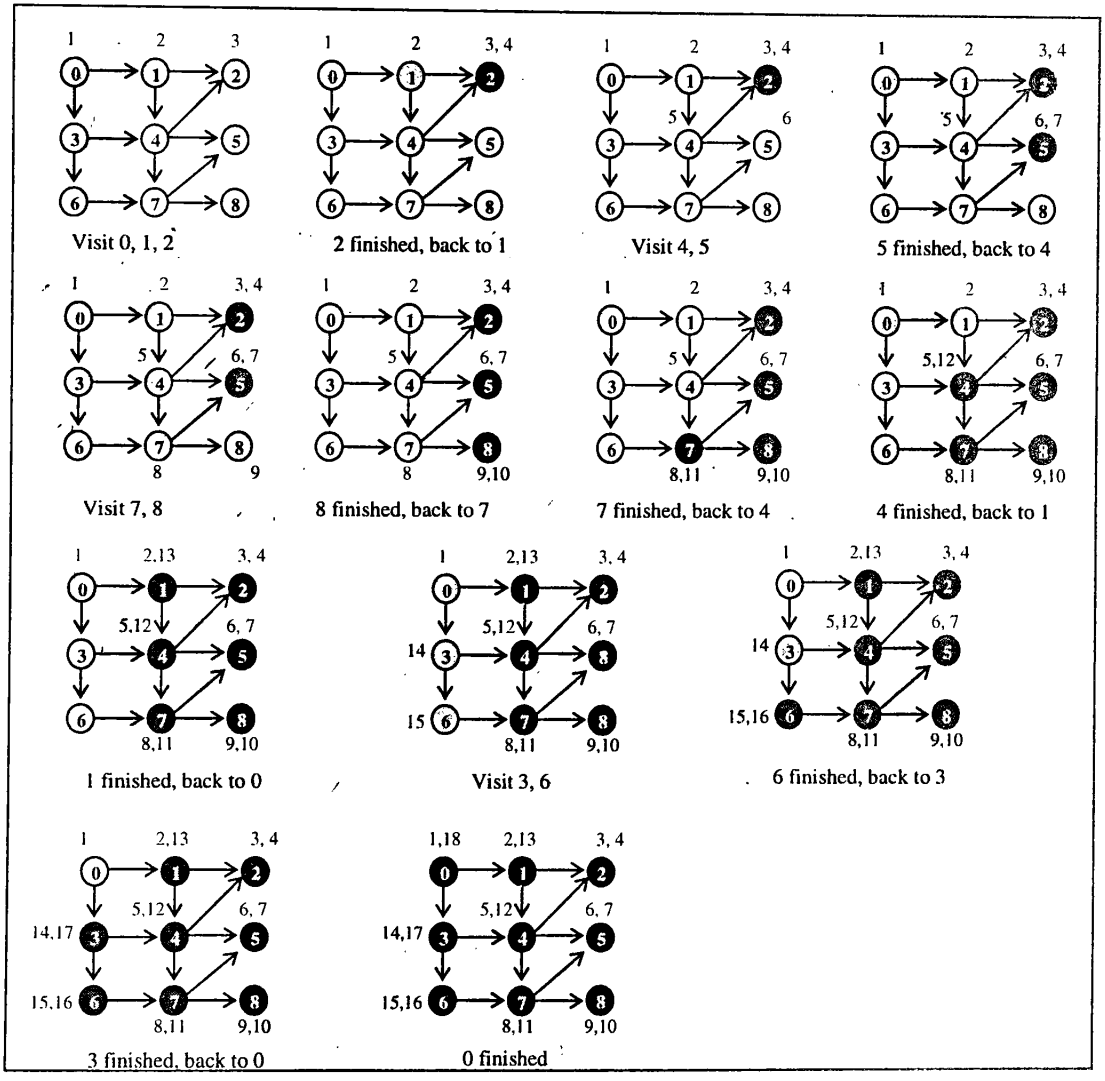

Visit 0, 1, 2
2 finished, back to 1
Visit 4, 5
5 finished, back to 4
Visit 7, 8
8 finished, back to 7
7 finished, back to 4
4 finished, back to 1
1 finished, back to 0
Visit 3, 6
6 finished, back to 3
3 finished, back to 0
0 finished

Figure 7.42 Recursive DFS showing starting time and finishing time

The recursive calls to DFS for the given graph would be-
(Ad means adjacent and In means initial)

```
►Call DFS(0) , visit 0
►1 is Ad to 0 and In : ■ Call DFS(1), visit 1
                       ■ 2 is Ad to 1 and In : - Call  DFS(2) , visit 2
                                               - Vertex 2 finished
                       ■ 4 is Ad to 1 and In : • Call DFS(4) , visit 4
                                               • 5 is Ad to 4 and In : - Call DFS(5) , visit 5
                                                                       - Vertex 5 finished
                                               • 7 is Ad to 4 and In : * Call  DFS(7) , visit 7
                                                                       * 8 is Ad to 7 and In :- Call DFS(8) , visit 8
                                                                                               -Vertex 8 finished
                                                                       * Vertex 7 finished
                                               • Vertex 4 finished
                       ■ Vertex 1 finished
►3 is Ad to 0 and In : ♦ Call DFS(3) , visit 3
                       ♦ 6 is Ad to 3 and In : - Call DFS(6) , visit 6
                                               - Vertex 6 finished
                       ♦ Vertex 3 finished
►Vertex 0 finished
```

If we number all these steps then we can get the discovery time and finishing time of all the vertices.

```
1. Call DFS(0) , visit 0
2.  1 is Ad to 0 and In :  Call DFS(1) , visit 1
                      3. 2 is Ad to 1 and In : Call  DFS(2) , visit 2
                                               4. Vertex 2 finished
                      5. 4 is Ad to 1 and In :  Call DFS(4) , visit 4
                                               6.  5 is Ad to 4 and In :  Call DFS(5) , visit 5
                                                                         7. Vertex 5 finished
                                               8.  7 is Ad to 4 and In :  Call  DFS(7) , visit 7
                                                                         9.  8 is Ad to 7 and In : Call DFS(8) , visit 8
                                                                                                  10.Vertex 8 finished
                                                                             11. Vertex 7 finished
                                               12. Vertex 4 finished
                      13. Vertex 1 finished
14. 3 is Ad to 0 and In :  Call DFS(3) , visit 3
                      15.  6 is Ad to 3 and In : Call DFS(6) , visit 6
                                               16. Vertex 6 finished
                      17. Vertex 3 finished
18. Vertex 0 finished
```

We can make simple additions in our recursive algorithm for recording the discovery time and finishing time of each vertex. We will take a variable `time` and initialize it to 0, and increment it whenever we visit a vertex or finish a vertex. The discovery times and finishing times are stored in arrays `d` and `f` respectively.

```
void DFS(int v)
{
        int i;
        time++;
        d[v] = time;    /*discovery time*/
        state[v] = visited;
        printf("%d ", v);
        for(i=0; i<n; i++)
        {
                if(adj[v][i]==1)
                {
                        if(state[i]==initial)
                                DFS(i);
                }
        }
        state[v] = finished;
        f[v] = ++time;  /*Finishing time*/
}/*End of DFS()*/
```

7.12.2.3 Classification of Edges in DFS

DFS can be used to classify the edges of a directed graph into four different groups. The edges can be divided into these groups based on the spanning forest.

(i) Tree edge - An edge included in the DFS spanning forest.
(ii) Back edge - An edge from a vertex to its spanning tree ancestor.
(iii) Forward edge - An edge from a vertex to a spanning tree non son descendant.
(iv) Cross edge - All remaining edges are cross edges. An edge between two vertices u and v is a cross edge, when there is no ancestor or descendent relationship between u and v in the spanning forest. The cross edges can be between the vertices of same spanning tree or vertices of different spanning trees.

Let us take a directed graph and apply DFS on it taking 0 as the start vertex.

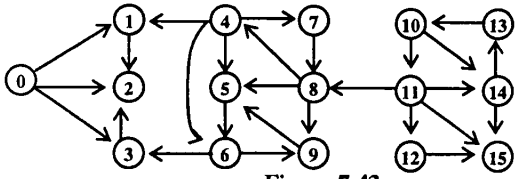


Figure 7.43

Vertices 1, 2, 3 are reachable from 1, so these are visited and then we select 4 as the next start vertex. The vertices 5, 6,7 8,9 are reachable from 4 so these are visited and then we select 10 as the next start vertex. All the remaining vertices are reachable from 10. The traversal would be -

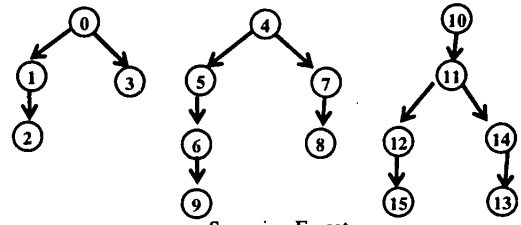
0, 1, 2, 3, 4, 5, 6, 9, 7, 8, 10, 11, 12, 15, 14, 13

The DFS spanning forest for the graph of figure 7.43 and different types of edges are shown in the figure 7.44.

The following function `DFS()` classifies all the edges in a directed graph.

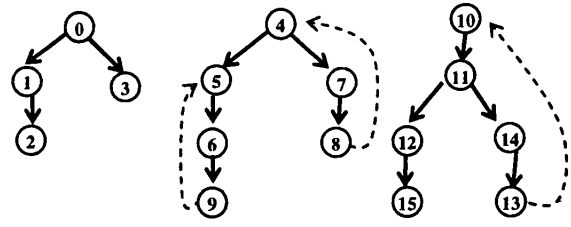
```
void DFS(int v)
{
        int i;
        time++;
        d[v] = time;
        state[v] = visited;
        for(i=0; i<n; i++)
        {
                if(adj[v][i]==1)
                {
                        if(state[i]==initial)
                        {
                                printf("(%d,%d) - Tree edge\n",v,i);
                                DFS(i);
                        }
                        else if(state[i]==visited)
                        {
                                printf("(%d,%d) - Back edge\n",v,i);
                        }
                        else if(d[v]<d[i])
                        {
                                printf("(%d,%d) - Forward edge\n",v,i);
                        }
                        else
                                printf("(%d,%d) - Cross edge\n",v,i);
                }
        }
        state[v] = finished;
        f[v] = ++time;
}/*End of DFS()*/
```

Tree edges
(0, 1), (1, 2), (0, 3), (4, 5), (5, 6),
(6, 9), (4, 7), (7, 8), (10, 11), (11, 12),
(12, 15), (11, 14), (14, 13)



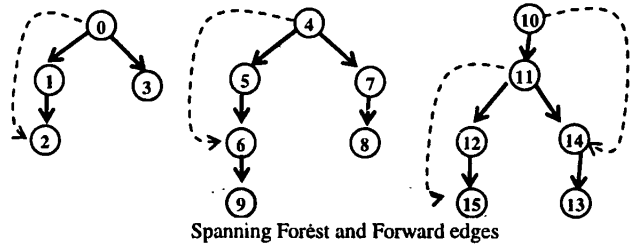
Spanning Forest

Back Edges
(9, 5), (8, 4), (13, 10)



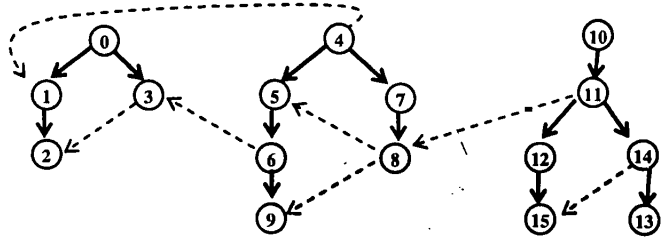
Spanning Forest and Back edges

Forward Edges
(0, 2), (4, 6), (11, 15), (10, 14)



Spanning Forest and Forward edges

Cross Edges
(3, 2), (4, 1), (6, 3), (8, 5),
(8, 9), (11, 8), (14, 15)



Spanning Forest and Cross edges

Figure 7.44

Now let us take an undirected graph and perform depth first search on it.

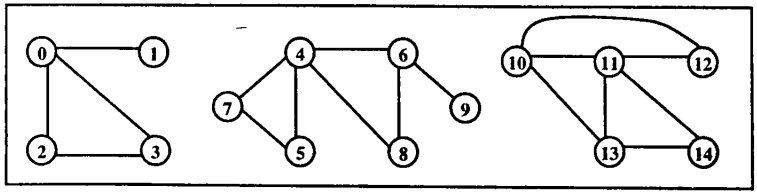


Figure 7.45

The depth first traversal of this graph would be-

0 1 2 3 4 5 7 6 8 9 10 11 12 13 14

The spanning forest for this graph would be-

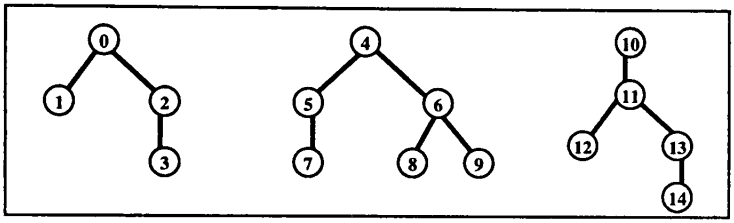


Figure 7.46

In an undirected graph, there is no difference between back edges and forward edges because all edges are bidirectional. So in an undirected graph, any edge between a vertex and its non son descendant is a back edge. Cross edges are also not possible in the depth first search of an undirected graph. Therefore in the depth first search of an undirected graph, every edge is classified as either tree edge or back edge. The following figure shows the tree edges and back edges for the undirected graph given above.

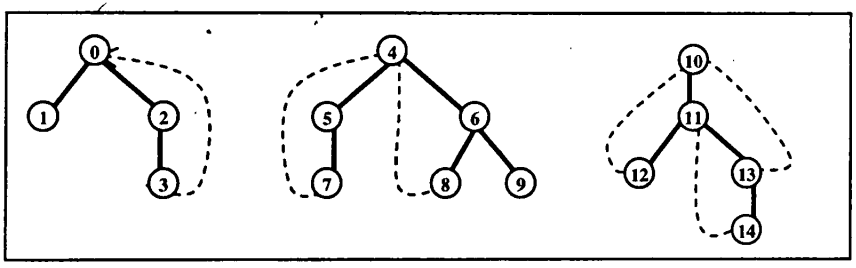


Figure 7.47

From this figure it is clear that in an undirected edge there can be no distinction between a forward edge and back edge so all edges from a vertex to non son descendants are back edges. Now let us see why cross edges are not possible. Suppose there was an edge in the graph from 12 to 14, then it seems to be a candidate for a cross edge looking at this spanning forest. But had the edge (12, 14) existed in the graph, the last spanning tree would have been different. In that case, 14 would be visited after 12 and so 14 would be the son of 12. Hence we can't have cross edges in a spanning tree of the forest. Similarly we can't have cross edges between different spanning trees. The following function DFS()shows how to classify the tree edges and back edges in an undirected graph.

```
void DFS(int v)
{
        int i;
        state[v] = visited;
        for(i=0; i<n; i++)
        {
                if(adj[v][i]==1 && predecessor[v]!=i)
                {
                        if(state[i]==initial)
                        {
                                predecessor[i] = v;
                                printf("(%d,%d) Tree edge\n",v,i);

                                DFS(i);
                        }
                        else if(state[i]==visited)
                        {
                                printf("(%d,%d) Back edge\n", v,i);
```

```
                    }
                }
            }
            state[v] = finished;
}/*End of DFS()*/
```

In an undirected graph, an edge between two vertices v1 and v2 is considered twice during DFS, once from v1 with v2 as its adjacent vertex, and once from v2 with v1 as its adjacent vertex. This can cause confusion in classifying the edges. To avoid this confusion, an edge between v1 and v2 is classified according to whether (v1,v2) or (v2,v1) is encountered first during the traversal. For example if there is an edge between vertices 5 and 8 of a graph. Now if vertex 5 is visited first then we will call edge (5,8) as tree edge. Vertex 5 will be made the predecessor of vertex 8. Now when vertex 8 will be visited, we will not consider edge (8,5). This is because we know that this edge has already been considered since 5 is predecessor of 8.

Depth first search can be used to find out whether a graph is cyclic or not. In both directed and undirected graphs, if we get a back edge during depth first traversal then the graph is cyclic.

7.12.2.4 Strongly connected graph and strongly connected components

In the section 7.5 we read about strongly connected graph and strongly connected components. Now we will apply depth first search to find whether a graph is strongly connected or not and to find the strongly connected components of a graph.

To prove that a graph G is strongly connected, we need to prove that these two statements are true for any vertex v of graph G.

1. All vertices of G are reachable from v, i.e. there is a path from v to every vertex.
2. v is reachable from all other vertices of G, there is a path from every vertex to v.

Now let us see why proving these two statements true is enough to prove that the graph is strongly connected. Let u and w be two vertices in graph G. To prove that G is strongly connected we need to show that there is a path from u to w and from w to u. If statements 1 and 2 are true, then we can show that these paths will exist.

For going from u to w, we can go from u to v(by st. 2) and then from v to w(by st. 1). Similarly for going from w to v, we can go from w to v(by st. 2) and then from v to u(by st.1). So to prove that a graph is strongly connected we need to prove that the 2 statements given above are true. We can perform `DFS(v)` and if it visits all the vertices, then we can say that first statement is true.

For proving the second statement we need to reverse the graph G, i.e. we have to reverse the direction of all the edges in G. Suppose G^R is the reverse graph, now we will perform `DFS(v)` on G^R, and if it visits all the vertices then the second statement is also true. The whole procedure is-

Take any vertex v of the graph G

1. Perform depth first search on G, starting at vertex v.
2. If it does not visit all the vertices
   Graph is not strongly connected, Return
   Else
   Reverse the graph G to get graph G^R
3. Perform Depth first search on G^R, starting at vertex v.
4. If it does not visit all the vertices
   Graph is not strongly connected.
   Else
   Graph is strongly connected.

Now let us see how we can find the strongly connected components (SCC) of a graph.

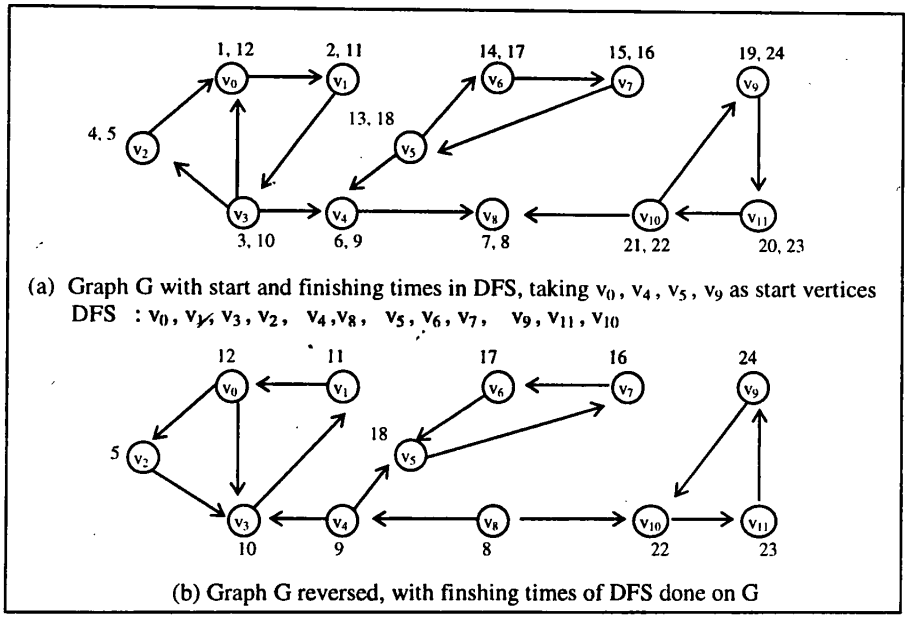
Take any vertex v of the graph G

1. Perform depth first search on G starting at vertex v, and mark the finishing time of each vertex.
2. Reverse the graph G to get graph G^R

3. Perform Depth First Search on G^R, always picking the vertex with highest finishing time for a new DFS.

The depth first trees in the depth first forest thus formed are the strongly connected components of G.

While performing DFS on the reverse graph, we start from the vertex that has the highest finishing time; all vertices reachable from this vertex will be in the same SCC. Now from the remaining vertices we pick the vertex that has highest finishing time and start DFS from there, all vertices reachable from this vertex will be in the same SCC. This process continues till all the vertices are visited. Let us take a graph and find strongly connected components for it.



(a) Graph G with start and finishing times in DFS, taking v_0, v_4, v_5, v_9 as start vertices
DFS : $v_0, v_1, v_3, v_2, v_4, v_8, v_5, v_6, v_7, v_9, v_{11}, v_{10}$

(b) Graph G reversed, with finshing times of DFS done on G

Figure 7.48

In figure 7.48(a), the graph is shown with the start and finishing time of each vertex, when DFS is done with v_0, v_4, v_5, v_9 as start vertices. In figure 7.48(b), the graph G is reversed and the finishing time from (a) is shown for each vertex.

To find strongly connected components, we will start DFS of the reverse graph from vertex v_9 since it had the highest finishing time in previous DFS. The vertices v_9, v_{10}, v_{11} are visited so they form one SCC. Now from the remaining vertices v_5 has the highest finishing time so we start DFS from there. The vertices v_5, v_7, v_6 are visited so they form another SCC. Now from the remaining vertices v_0 has the highest finishing time so we start DFS from there. The vertices v_0, v_2, v_3, v_1 are visited so they form another SCC. Next v_4 has highest finishing time so DFS starts from there and only v_4 is visited and so it forms one SCC. The remaining vertex v_8 also forms a SCC. The strongly connected components are-

$\{v_9, v_{10}, v_{11}\}, \{v_5, v_7, v_6\}, \{v_0, v_2, v_3, v_1\}, \{v_4\}, \{v_8\}$

7.13 Shortest Path Problem

There can be several paths possible for going from one vertex to another vertex in a weighted graph, but the shortest path is the path in which the sum of weights of the included edges is the minimum. Consider the weighted directed graph in figure 7.49.

Suppose we have to go from vertex 0 to vertex 4. We can take the path 0→2→3→4 and the length of this path would be 16. But this is not the only path from vertex 0 to vertex 4, there are other paths which may be shorter. We are interested in finding the shortest of these paths. The other two paths are 0→2→5→3→4 length=12) 0→1→4 (length=11). So from all the three paths, the shortest one is 0→1→4. It is not necessary that the shortest path is unique.

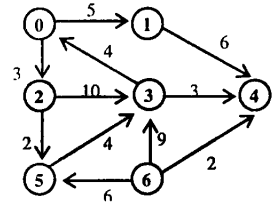


Figure 7.49

We will study three algorithms for finding out the shortest paths in a weighted graph.

(i) Dijkstra's Algorithm - Single source, non negative weights.

(ii) Bellman Ford Algorithm - Single source, general weights.

(iii) Floyd's or Modified Warshall's algorithm - All pairs shortest paths.

The first two algorithms are single source shortest paths algorithms (this source is different from the 0 indegree source defined in section 7.3). In these shortest path problems, a vertex is identified as the source vertex, and shortest path from this vertex to all other vertices is found out. This source vertex can also be called the start vertex. Dijkstra's algorithm works only for non negative weights while Bellman Ford algorithm can be used for negative weights also.

There is no algorithm that finds shortest path from source to a single destination, and is faster than the single source shortest paths algorithms. So finding shortest paths from a single source to all vertices is as simple as finding shortest path to a single destination.

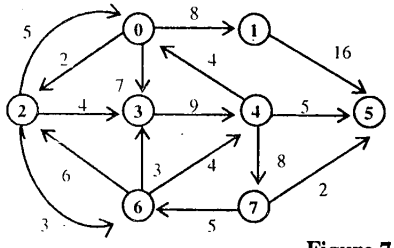
The Floyd's algorithm is an all pairs shortest path problem. Here we get the shortest paths between each pair of vertices of the graph. This is also called Modified Warshall's algorithm because it is based on the Warshall's algorithm that we studied earlier in section 7.11.2.

The shortest paths can prove useful in various situations. For example suppose our weighted graph represents a transport system, where each vertex is a city and weights on the edges represent the distance of one city from another. We would be interested in taking the shortest route to reach our destination. Similarly the graph may represent an airlines network or a railway track system and the weights on the edges may represent the distance, time or cost. In all these cases, the knowledge of shortest paths helps us select the best option. Electric supply system and water distribution system also follow this approach. In computers, it is very useful in network for routing concepts.

7.13.1 Dijkstra's Algorithm

Dijkstra's algorithm is named after E. W. Dijkstra who gave this technique of finding shortest paths in 1959. In this algorithm we find the shortest distance from source vertex to all other vertices in the graph. This algorithm works correctly only if all the weights in the graph are nonnegative.

Consider the graph given in figure 7.50. If we assume 0 as the source vertex then the table next to it shows the shortest paths to all other vertices of the graph.



Source	Destination	Shortest Path	Length
0	1	0-1	8
0	2	0-2	2
0	3	0-2-3	6
0	4	0-2-6-4	9
0	5	0-2-6-4-5	14
0	6	0-2-6	5
0	7	0-2-6-4-7	17

Figure 7.50

Now we will see how we can obtain these shortest paths using Dijkstra's algorithm. Each vertex is given a status, which can be permanent or temporary. If a vertex is temporary, then it means that shortest path to it has not been found and if a vertex is permanent then it means shortest path to it has been found. Initially all the vertices are temporary and at each step of the algorithm, a temporary vertex is made permanent.

We label each vertex with pathLength and predecessor. At any point of the algorithm, the pathLength of a vertex v will denote the length of shortest path – known till now – from source vertex to v. The predecessor of v will denote the vertex which precedes v in this path.

Initially the pathLength of all vertices are initialized to a very large number which denotes that at the start of algorithm we don't know of any path from source to any vertex. We will call this number as infinity. The predecessor value of all vertices is initialzed to NIL. In the program we can take NIL to be any number that does not represent a valid vertex. We will take it -1 in our program since our vertices start from 0.

At the end of the algorithm, pathLength of a vertex will represent the shortest distance of that vertex from the source vertex and predecessor will represent the vertex which precedes the given vertex in the shortest path from source.

As the algorithm proceeds, the values of pathLength and predecessor of a vertex may be updated many times provided the vertex is temporary. Once a vertex is made permanent the values of pathLength and predecessor for it become fixed and are not changed thereafter. It means that temporary vertices can be relabeled if required, but permanent vertices can't be relabeled.

When a temporary vertex is made permanent, it means that the shortest distance for it has been finalised. So pathLength of a permanent vertex represents the length of shortest path from source to this vertex, i.e. no other path shorter than this is possible.

At any point of the algorithm, the pathLength of a temporary vertex represents the length of best known path – from source to this vertex– till now, it is possible that there may be some other better path which is shorter than this one. So whenever we will find a shorter path we will update the pathLength and predecessor values of this temporary vertex. We will try to find shorter paths by examining the edges incident from the vertex which is made permanent most recently.

We have stated that at each step a vertex will be made permanent, now the question is which vertex should be chosen to become permanent. For this we use the greedy approach. In greedy algorithms we generally perform an action which appears best at the moment. Greedy algorithms do not always give optimal results in general, but in this case we get the correct result and we will see the proof at the end. Applying the greedy approach, in each step the temporary vertex that has the smallest value of pathLength is made permanent.

Now let us look at the whole algorithm stepwise and see how the pathLength and predecessor values are updated to get shorter paths and how we finally get the shortest paths for all the vertices. The procedure is-

(A) Initialize the pathLength of all vertices to infinity and predecessor of all vertices to NIL. Make the status of all vertices temporary.

(B) Make the pathLength of source vertex equal to 0.

(C) From all the temporary vertices in the graph, find out the vertex that has minimum value of pathLength, make it permanent and now this is our current vertex.(If there are many with the same value of pathLength then anyone can be selected).

(D) Examine all the temporary vertices adjacent to the current vertex. The value of pathLength is recalculated for all these temporary successors of current, and relabelling is done if required. Let us see how this is done.

Suppose `s` is the source vertex, `current` is the current vertex and `v` is a temporary vertex adjacent to `current`.

(i) If pathLength(`current`) + weight(`current`, v) < pathLength(v)

It means that the path from `s` to `v` via `current` is smaller than the path currently assigned to `v`. We've found a shorter path so pathLength(v) is updated and assigned the length of this smaller path. Also the predecessor of `v` is changed to `current` since now `current` is the predecessor of the vertex `v` in the shortest path. So in this case the vertex `v` is relabelled. Relabelling means discard the previous path from `s` to v, and update it to a new path which consists of a path from `s` to `current` and a direct edge from `current` to `v`.

Now pathLength(v) = pathLength(`current`) + weight(`current`, v)
predecessor(v) = current

(ii) If pathLength(current) + weight(current, v) >= pathLength(v)

It means that using current vertex in the path from s to v does not offer any shorter path. So in this case vertex v is not relabelled and values of pathLength and predecessor for vertex v remain unchanged.

(E) Repeat steps C and D until there is no temporay vertex left in the graph, or all the temporary vertices left have pathLength of infinity.

Now we will take the graph given in figure 7.50 and find out the shortest paths taking 0 as the source vertex.

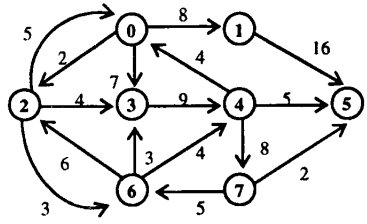
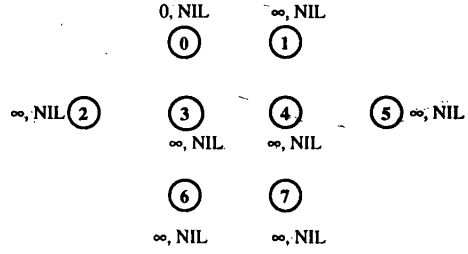


Figure 7.51

Initially no paths are known so the pathLength values for all the vertices are set to a very large number(which is larger than length of longest possible path), we will represent this by ∞ in the figure. The predecessor of all vertices is NIL in the beginning. Source vertex 0 is assigned a pathLength of 0. In the figures, all vertices will be joined to their predecessors. A permanent vertex will be joined to its predecessor by a bold arrow.



Vertex	path Length	predecessor	status
0	0	NIL	Temp
1	∞	NIL	Temp
2	∞	NIL	Temp
3	∞	NIL	Temp
4	∞	NIL	Temp
5	∞	NIL	Temp
6	∞	NIL	Temp
7	∞	NIL	Temp

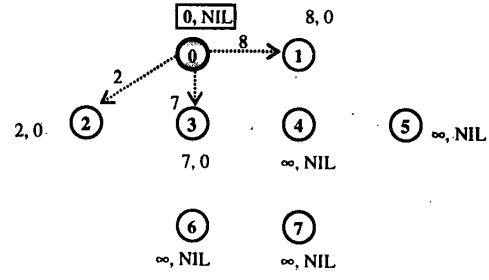
From all the temporary vertices, vertex 0 has the smallest pathLength, so make it permanent. Its predecessor will remain NIL. Now vertex 0 is the current vertex.

Vertices 1, 2, 3 are temporary vertices adjacent to vertex 0.

pathLength(0) + weight(0,1) < pathLength(1) 0+8 < ∞
Relabel 1, pathLength[1] = 8, predecessor[1] = 0

pathLength(0) + weight(0,2) < pathLength(2) 0+2 < ∞
Relabel 2, pathLength[2] = 2, predecessor[2] = 0

pathLength(0) + weight (0,3) < pathLength(3) 0+7 < ∞
Relabel 3, pathLength[3] = 7, predecessor[3] = 0

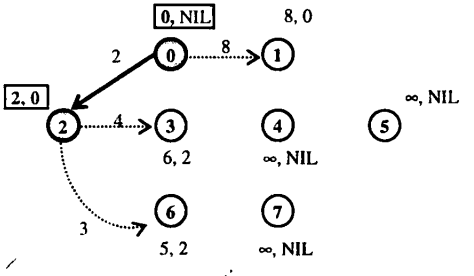


Vertex	path Length	prede-cessor	status
0	0	NIL	Perm
1	8	0	Temp
2	2	0	Temp
3	7	0	Temp
4	∞	NIL	Temp
5	∞	NIL	Temp
6	∞	NIL	Temp
7	∞	NIL	Temp

From all the temporary vertices, vertex 2 has the smallest pathLength so make it permanent. Now vertex 2 is the current vertex.
Vertices 3, 6 are temporary vertices adjacent to vertex 2.

pathLength(2) + weight(2,3) < pathLength(3) 2+4 < 7
Relabel 3, pathLength[3] = 6, predecessor[3] = 2

pathLength(2) + weight(2,6) < pathLength(6) 2+3< ∞
Relabel 6, pathLength[6] = 5, predecessor[6] = 2

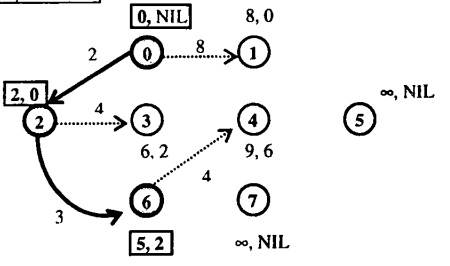


Vertex	path Length	prede-cessor	status
0	0	NIL	Perm
1	8	0	Temp
2	2	0	Perm
3	6	2	Temp
4	∞	NIL	Temp
5	∞	NIL	Temp
6	5	2	Temp
7	∞	NIL	Temp

From all the temporary vertices, vertex 6 has smallest pathLength so make it permanent. Now vertex 6 is our current vertex .Vertices 3, 4 are temporary vertices adjacent to vertex 6.

pathLength(6) + weight(6,3) > pathLength(3) 5+3 > 6
Don't relabel 3

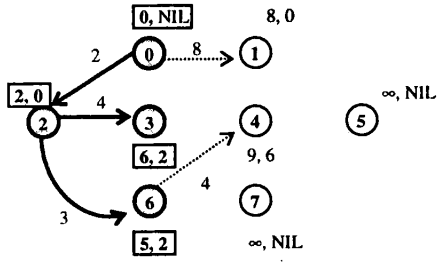
pathLength(6) + weight(6,4) < pathLength(4) 5+4 < ∞
Relabel 4, pathLength[4] = 9, predecessor[4] = 6



Vertex	path Length	prede-cessor	status
0	0	NIL	Perm
1	8	0	Temp
2	2	0	Perm
3	6	2	Temp
4	9	6	Temp
5	∞	NIL	Temp
6	5	2	Perm
7	∞	NIL	Temp

From all temporary vertices, vertex 3 has smallest pathLength so make it permanent. Now vertex 3 is the current vertex.
Vertex 4 is the temporary vertex adjacent to 3.

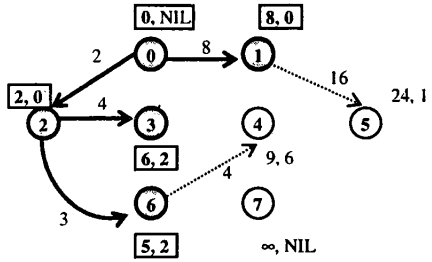
pathLength(3) + weight(3,4) > pathLength(4) 6+9 > 9
Don't relabel 4



Vertex	path Length	prede -cessor	status
0	0	NIL	Perm
1	8	0	Temp
2	2	0	Perm
3	6	2	Perm
4	9	6	Temp
5	∞	NIL	Temp
6	5	2	Perm
7	∞	NIL	Temp

From all temporary vertices, vertex 1 has smallest pathLength so make it permanent. Now vertex 1 is the current vertex. Vertex 5 is the temporary vertex adjacent to 1.

pathLength(1) + weight(1,5) < pathLength(5) 8+16 < ∞
Relabel 5, pathLength[5] = 24, predecessor[5] = 1

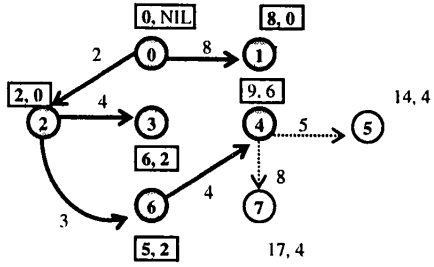


Vertex	path Length	prede-cessor	status
0	0	NIL	Perm
1	8	0	Perm
2	2	0	Perm
3	6	2	Perm
4	9	6	Temp
5	24	1	Temp
6	5	2	Perm
7	∞	NIL	Temp

Fom all temporary vertices, vertex 4 has smallest pathLength, so make it permanent. Now vertex 4 is the current vertex. Vertices 5 and 7 are temporary vertices adjacent to 4.

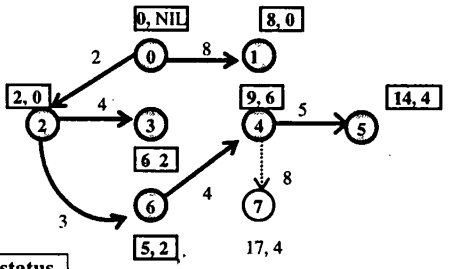
pathLength(4) + weight(4,5) < pathLength(5) 9+5 < 24
Relabel 5, pathLength[5] = 14, predecessor[5] = 4

pathLength(4) + weight(4,7) < pathLength(7) 9+8 < ∞
Relabel 7, pathLength[7] = 17, predecessor[7] = 4



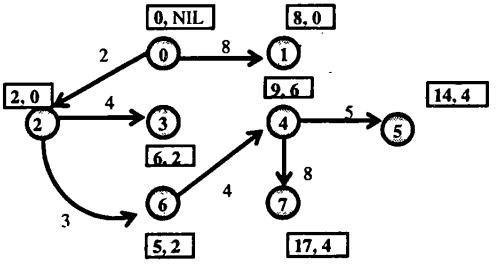
Vertex	path Length	prede -cessor	status
0	0	NIL	Perm
1	8	0	Perm
2	2	0	Perm
3	6	2	Perm
4	9	6	Perm
5	14	4	Temp
6	5	2	Perm
7	17	4	Temp

From all temporary vertices, vertex 5 has smallest pathLength so make it permanent. Now vertex 5 is the current vertex. There is no vertex adjacent to 5.



Vertex	path Length	prede -cessor	status
0	0	NIL	Perm
1	8	0	Perm
2	2	0	Perm
3	6	2	Perm
4	9	6	Perm
5	14	4	Perm
6	5	2	Perm
7	17	4	Temp

Now 7 is the only temporary vertex left and it has pathLength 17, so make it permanent.



Vertex	path Length	prede-cessor	status
0	0	NIL	Perm
1	8	0	Perm
2	2	0	Perm
3	6	2	Perm
4	9	6	Perm
5	14	4	Perm
6	5	2	Perm
7	17	4	Perm

Now all the vertices have become permanent.

At the end we get a shortest path tree which includes all the vertices reachable from the source vertex. The source vertex is the root of this tree and the shortest paths from source to all vertices are given by the branches. Each vertex has a predecessor so we can easily establish the path after completing the whole process. To find the shortest path from source to any destination vertex, we look at the last table, start from the destination vertex and keep on following successive predecessors till we reach the source vertex.

If destination vertex is 3
predecessor of 3 is 2, predecessor of 2 is 0
Shortest Path is 0 - 2 - 3

If destination vertex is 5
predecessor of 5 is 4, predecessor of 4 is 6, predecessor of 6 is 2, predecessor of 2 is 0
Shortest path is 0 - 2 - 6 - 4 - 5

If we want to find the shortest distance between source and a single destination only, then we can stop our algorithm as soon as the destination vertex is made permanent.

In the example that we have taken, all the vertices of the graph were reachable from the source vertex, so all the vertices were permanent in the last table. If there are one or more vertices in the graph that are not reachable from the source vertex, then the shortest path to all these vertices cannot be found so they will never become permanent. For example in the graph given below the vertices 1 and 7 are not reachable from source vertex 0.

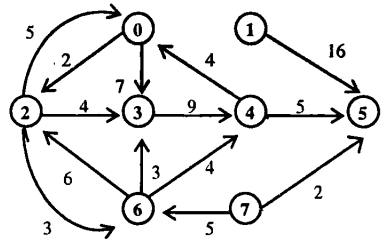


Figure 7.52

Following the same procedure, vertices 0, 2, 6, 3, 4, 5 will be made permanent and then the pathLength and predecessor values of all vertices will be-

Vertex	path Length	prede-cessor	status
0	0	NIL	Perm
1	∞	NIL	Temp
2	2	0	Perm
3	6	2	Perm
4	9	6	Perm
5	14	4	Perm
6	5	2	Perm
7	∞	NIL	Temp

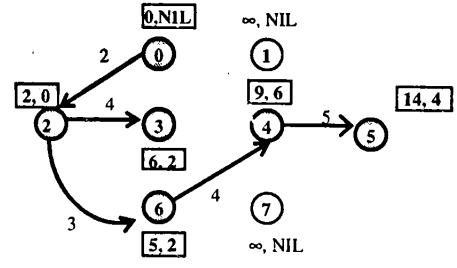


Figure 7.53

Now all the temporary vertices left have pathLength equal to infinty. While stating the procedure of Dijkstra's algorithm we had mentioned that the procedure will stop when either no temporary vertices are left or all temporary vertices left have pathLength equal to infinity. So in this example we will stop after making vertex 5 permanent and the vertices 1 and 7 will never be made permanent. These vertices are not reachable from the source vertex.

```
/*P7.9 Program to find shortest distances using Dijkstra's algorithm*/
#include<stdio.h>
#define MAX 100
#define TEMP 0
#define PERM 1
#define infinity 9999
#define NIL -1

void findPath(int s,int v);
void Dijkstra(int s);
int min_temp();
void create_graph();

int n;     /*Denotes number of vertices in the graph*/
int adj[MAX][MAX];
int predecessor[MAX];  /*predecessor of each vertex in shortest path*/
int pathLength[MAX];
int status[MAX];

main()
{
        int s,v;

        create_graph();
        printf("Enter source vertex : ");
        scanf("%d",&s);
        Dijkstra(s);

        while(1)
        {
                printf("Enter destination vertex(-1 to quit): ");
                scanf("%d",&v);
                if(v==-1)
                   break;
                if(v<0 || v>=n)
                   printf("This vertex does not exist\n");
                else if(v==s)
                   printf("Source and destination vertices are same\n");
                else if(pathLength[v]==infinity)
                 printf("There is no path from source to destination vertex\n");
                else
                   findPath(s,v);
        }
}/*End of main()*/
void Dijkstra(int s)
{
        int i,current;

        /*Make all vertices temporary*/
        for(i=0; i<n; i++)
        {
                predecessor[i] =  NIL;
                pathLength[i] = infinity;
                status[i] = TEMP;
        }

        /*Make pathLength of source vertex equal to 0*/
        pathLength[s] = 0;
        while(1)
        {
           ;    /*Search for temporary vertex with minimum pathLength
                and make it current vertex*/
                current = min_temp();
                if(current==NIL)
                        return;
                status[current] = PERM;
```

```
                for(i=0; i<n; i++)
                {
                 /*Checks for adjacent temporary vertices*/
                  if(adj[current][i]!=0 && status[i]==TEMP)
                   if(pathLength[current] + adj[current][i] < pathLength[i])
                   {
                        predecessor[i] = current;  /*Relabel*/
                        pathLength[i] = pathLength[current] + adj[current][i];
                   }
                }
        }
}/*End of Dijkstra()*/
/*Returns the temporary vertex with minimum value of pathLength, Returns NIL if no temporary
vertex left or all temporary vertices left have pathLength infinity*/
int min_temp()
{
        int i;
        int min = infinity;
        int k = NIL;
        for(i=0; i<n; i++)
        {
                if(status[i]==TEMP && pathLength[i]<min)
                {
                        min = pathLength[i];
                        k = i;
                }
        }
        return k;
}/*End of min_temp()*/
void findPath(int s,int v)
{
        int i,u;
        int path[MAX]; /*stores the shortest path*/
        int shortdist = 0;     /*length of shortest path*/
        int count = 0; /*number of vertices in the shortest path*/
        /*Store the full path in the array path*/
        while(v!=s)
        {
                count++;
                path[count] = v;
                u = predecessor[v];
                shortdist += adj[u][v];
                v = u;
        }
        count++;
        path[count]=s;
        printf("Shortest Path is : ");
        for(i=count; i>=1; i--)
                printf("%d  ",path[i]);
        printf("\n Shortest distance is : %d\n", shortdist);
}/*End of findPath()*/
void create_graph()
{
        int i,max_edges,origin,destin,wt;
        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edges = n*(n-1);
        for(i=1; i<=max_edges; i++)
        {
                printf("Enter edge %d(-1 -1 to quit) : ",i);
                scanf("%d %d",&origin,&destin);
```

```
            if((origin==-1) && (destin==-1))
                    break;
            printf("Enter weight for this edge : ");
            scanf("%d",&wt);
            if(origin>=n || destin>=n || origin<0 || destin<0)
            {
                    printf("Invalid edge!\n");
                    i--;
            }
            else
                    adj[origin][destin] = wt;
    }
}
```

The function `create_graph()` is same as in program P7.2. Now after studying the whole procedure let us see why this algorithm works and gives the optimal result.

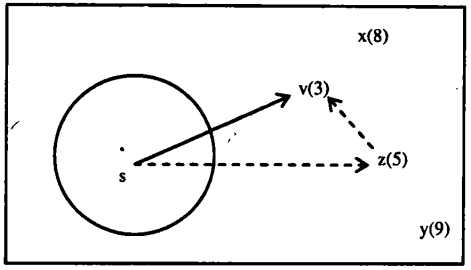


Figure 7.54

Suppose we have 4 temporary vertices x, y, z, v. Vertices inside circle are permanent vertices and vertices outside circle are temporary vertices. We take the vertex with smallest pathLength and declare that shortest path to it has been finalized and make it permanent. In this case v has shortest value of pathLength so it is declared permanent. Let us see why this is the shortest path for v.

If this is not the shortest path for v, then suppose there exists a hypothetical shorter path going through z. This path goes from s to z and then z to v. Now length of this path is equal to sum of pathLength(z) and weight(z,v). We are claiming that this path is shorter so the value pathLength(z)+weight(z, v) should be smaller than pathLength(v). If this is so then pathLength(z) will also be smaller than pathLength(v). Here comes the contradiction - if pathLength(z) is smaller than pathLength(v) then our greedy approach would have chosen z to be made permanent instead of v. So we have proved by contradiction that pathLength(v) is the shortest distance from s to v.

7.13.2 Bellman Ford Algorithm

Dijkstra's algorithm works properly only for non negative weights. For example consider the graph given below-

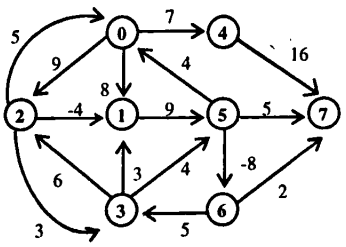


Figure 7.55

The shortest distances from the source vertex 0 to all other vertices as computed by Dijkstra's algorithm are-

Source	Destination	Shortest Path	Length
0	1	0-1	8
0	2	0-2	9
0	3	0-2-3	12
0	4	0-4	7
0	5	0-2-3-5	16
0	6	0-2-3-5-6	8
0	7	0-2-3-5-6-7	10

Figure 7.56

Using Dijkstra's algorithm, the shortest path from 0 to 1 is the path 0→1 of length 8. But if we observe the graph carefully we see that there exists a shorter path 0→2→1 of length 5. Similarly for destination vertex 5 there exists a path 0→2→1→5 of length 14 which Dijkstra's algorithm was unable to identify.

So we can see that Dijkstra's algorithm fails if the graph consists of negative weights. This is so because in Dijkstra's algorithm once a vertex is made permanent we don't relabel it i.e. the shortest path to it is finalized. It is possible that after making the vertex permanent we find an edge of negative weight that can be used to reach the vertex and hence we get a shorter path. But we have already finalized the shortest path of the vertex by making it permanent, so we can't record this shorter path.

In the graph of figure 7.55, if we apply Dijkstra's algorithm, then the vertices are made permanent in the order - 0 4 1 2 3 5 6 7.

After making vertex 2 permanent, the edge 2→1 of length -4 is considered and we find a shorter path 0→2→1 but it is not recorded, as vertex 1 is already permanent.

In Dijkstra's algorithm we make a vertex permanent at each step, i.e. the shortest distance to a vertex is finalized at each step but in Bellman Ford algorithm the shortest distances are not finalized till the end of the algorithm. Thus in Bellman-Ford algorithm, we drop the concept of making vertices permanent. This is why Dijkstra's algorithm is known as label setting algorithm and Bellman Ford algorithm is known as label correcting algorithm.

Each vertex is labeled with a pathLength and a predecessor value as in Dijkstra's algorithm. The procedure for Bellman Ford algorithm is-

(A) Initialize the pathLength of all vertices to infinity and predecessor of all vertices to NIL.

(B) Make the pathLength of source vertex equal to 0 and insert it into the queue.

(C) Delete a vertex from the front of the queue and make it the current vertex.

(D) Examine all the vertices adjacent to the current vertex. Check the condition of minimum weight for these vertices and do the relabeling if required, as in Dijkstra's algorithm.

(E) Each vertex that is relabeled is inserted into the queue provided it is not already present in the queue.

(F) Repeat the steps (C), (D), and (E) till the queue becomes empty.

The whole procedure for the graph of figure 7.55 is shown in the following table.

Current Vertex	Adjacent vertices		Queue
0	1, 2, 4	**1** 0+8 < ∞ pathLength(1) = 8, pred(1) = 0, Enqueue 1 **2** 0+9 < ∞ pathLength(2) = 9, pred(2) = 0, Enqueue 2 **4** 0+7 < ∞ pathLength(4) = 7, pred(4) = 0, Enqueue 4	1 2 4
1	5	**5** 8+9 < ∞ pathLength(5) = 17, pred(5) = 1, Enqueue 5	2 4 5
2	0, 1, 3	**0** 9+5 > 0 **1** 9 +(- 4) < 8 pathLength(1) =5, pred(1) = 2, Enqueue 1 **3** 9+3 < ∞ pathLength(3) =12,pred(3) = 2, Enqueue 3	4 5 1 3
4	7	**7** 7+16 < ∞ pathLength(7) =23, pred(7)= 4, Enqueue 7	5 1 3 7
5	0, 6, 7	**0** 17+4 > 0 **6** 17+(-8) < ∞ pathLength(6) =9, pred(6)=5, Enqueue 6 **7** 17+5 < 23 pathLength(7) =22, pred(7)= 5, 7 already in queue	1 3 7 6
1	5	**5** 5 +9 < 17 pathLength(5) =14, pred(5)=1, Enqueue 5	3 7 6 5
3	1, 2, 5	**1** 12+3 > 5 **2** 12 +6 > 9 **5** 12+4 > 14	7 6 5
7	-	-	6 5
6	3, 7	**3** 9+5 > 12 **7** 9+2 < 22 pathLength(7) =11, pred(7)= 6, Enqueue 7	5 7
5	0, 6,7	**0** 14+4>0 **6** 14+(-8) < 9 pathLength(6) =6, pred(6)= 5, Enqueue 6 **7** 14+5 > 11	7 6
7	-	-	6
6	3, 7	**3** 6+5 < 12 pathLength(3) =11, pred(3) = 6, Enqueue 3 **7** 6 +2 < 11 pathLength(7) = 8, pred(7) = 6, Enqueue 7	3 7
3	1, 2, 5	**1** 11+3 > 5 **2** 11+6 > 9 **5** 11+4 > 14	7
7	-	-	Empty

Figure 7.57

The shortest paths in this way computed by Bellman Ford algorithm for source vertex 0 are given below-

Source	Destination	Shortest Path	Length
0	1	0-2-1	5
0	2	0-2	9
0	3	0-2-1-5-6-3	11
0	4	0-4	7
0	5	0-2-1-5	14
0	6	0-2-1-5-6	6
0	7	0-2-1-5-6-7	8

Figure 7.58

This algorithm will not work properly if the graph contains a negative cycle reachable from source vertex i.e. a cycle consisting of edges whose weights add up to a negative number. For example consider the following graph which contains a negative cycle 0→1→3→0 of length -3.

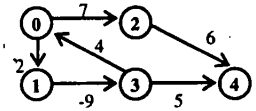


Figure 7.59

Now suppose we have to find the shortest path from vertex 0 to vertex 3. One path from vertex 0 to vertex 3 is 0→1→3 of length -7. But there is a shorter path which is 0→1→3→0→1→3 of length -10. Each time we visit the negative cycle the length of the shortest path will decrease by -3 and hence the length of shortest path from vertex 0 to vertex 3 is -∞. So if the graph contains a negative cycle then the number of edges in the shortest path is not finite and hence shortest paths are not defined for such graphs.

In the Bellman Ford algorithm each vertex can be inserted into the queue maximum n times. If any vertex is inserted more than n times, it indicates that there is negative cycle present in the graph. In this case we will be stuck in an infinite loop. To come out of it we can count the number of insertion of any vertex and if it is greater than n, we will come out of the loop stating that graph has a negative cycle. In the program we will count the number insertions of source vertex.

```
/*P7.10 Program to find shortest paths using Bellman-Ford algorithm*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define infinity 9999
#define NIL -1
#define TRUE 1
#define FALSE 0
int n;    /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Adjacency Matrix*/
int predecessor[MAX];
int pathLength[MAX];
int isPresent_in_queue[MAX];
int front,rear;
int queue[MAX];
void initialize_queue();
void insert_queue(int u);
int delete_queue();
int isEmpty_queue();
void create_graph();
void findPath(int s, int v);
int BellmanFord(int s);
main()
{
        int flag,s,v;
        create_graph();
        printf("Enter source vertex : ");
        scanf("%d",&s);
        flag = BellmanFord(s);
        if(flag==-1)
        {
                printf("Error : negative cycle in Graph\n");
                exit(1);
        }
        while(1)
        {
                printf("Enter destination vertex(-1 to quit): ");
                scanf("%d",&v);
                if(v==-1)
                  break;
                if(v<0 || v>=n)
                  printf("This vertex does not exist\n");
```

```
            else if(v==s)
              printf("Source and destination vertices are same\n");
            else if(pathLength[v]==infinity)
            printf("There is no path from source to destination vertex\n");
            else
              findPath(s,v);
        }
}/*End of main()*/
void findPath(int s, int v)
{
        int i,u;
        int path[MAX]; /*scores the shortest path*/
        int shortdist = 0;      /*length of shortest path*/
        int count = 0; /*number of vertices in the shortest path*/
        /*Store the full path in the array path*/
        while(v!=s)
        {
                count++;
                path[count] = v;
                u = predecessor[v];
                shortdist += adj[u][v];
                v = u;
        }
        count++;
        path[count]=s;
        printf("Shortest Path is : ");
        for(i=count; i>=1; i--)
                printf("%d  ",path[i]);
        printf("\n Shortest distance is : %d\n", shortdist);
}/*End of findPath()*/
int BellmanFord(int s)
{
        int k=0,i,current;
        for(i=0; i<n; i++)
        {
                predecessor[i] = NIL;
                pathLength[i] = infinity;
                isPresent_in_queue[i] = FALSE;
        }
        initialize_queue();
        pathLength[s] = 0; /*Make pathLength of source vertex 0*/
        insert_queue(s); /*Insert the source vertex in the queue*/
        isPresent_in_queue[s] = TRUE;
        while(!isEmpty_queue())
        {
                current = delete_queue();
                isPresent_in_queue[current] = FALSE;
                if(s==current)
                        k++;
                if(k>n)
                        return -1;/*Negative cycle reachable from source vertex*/
                for(i=0; i<n; i++)
                {
                  if(adj[current][i] != 0)
                    if(pathLength[i] > pathLength[current]+adj[current][i])
                    {
                        pathLength[i] = pathLength[current]+adj[current][i];
                        predecessor[i] = current;
                        if(!isPresent_in_queue[i])
                        {
                          insert_queue(i);
                          isPresent_in_queue[i]=TRUE;
                        }
                    }
```

```
                }
        }
        return 1;
}/*End of BellmanFord()*/
void initialize_queue()
{
        int i;
        for(i=0; i<MAX; i++)
                queue[i] = 0;
        rear = -1;front.= -1;
}/*End of initailize_queue()*/
int isEmpty_queue()
{
        if(front==-1 || front>rear)
                return 1;
        else
                return 0;
}/*End of isEmpty_queue()*/
void insert_queue(int added_item)
{
        if(rear==MAX-1)
        {
                printf("Queue Overflow\n");
                exit(1);
        }
        else
        {
                if(front==-1)  /*If queue is initially empty */
                        front = 0;
                rear = rear+1;
                queue[rear] = added_item ;
        }
}/*End of insert_queue()*/
int delete_queue()
{
        int d;
        if(front==-1 || front>rear)
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        else
        {
                d = queue[front];
                front = front+1;
        }
        return d;
}/*End of delete_queue()*/
```

7.13.3 Modified Warshall's Algorithm (Floyd's Algorithm)

We have studied two algorithms to find out the shortest paths; both of them were single source problems. Now the algorithm that we are going to study is an all pairs shortest path problem, i.e. it finds the shortest path between all pairs of vertices in the graph.

We may apply Dijkstra's algorithm n times, once for each vertex as the source vertex and find the shortest path between all pairs of vertices. The time complexity in this case would be $O(n^3)$, and the time complexity of our Floyd's algorithm is also $O(n^3)$, but the Floyd's algorithm is much simpler and is faster if the graph is dense. Moreover negative weights are also allowed in Floyd's algorithm. The only restriction is that graph should not have any negative cycle.

We have seen Warshall's algorithm(section 7.11.2) that computes the path matrix of a graph and tells us whether there is a path between any two vertices i and j. Now our problem is to find the shortest path between any two vertices i and j. We will take Warshall's algorithm as the base and modify it to find out the shortest path matrix D, such that D[i][j] represents the length of shortest path from vertex i to vertex j. This resulting algorithm is known as modified Warshall's algorithm or Floyd's algorithm or Floyd Warshall algorithm as its basic structure was given by Warshall and it was implemented by Robert. W. Floyd. Any element D[i][j] of the shortest path matrix can be defined as-

$$[i][j] = \begin{cases} \text{Length of shortest path from vertex i to vertex j} \\ \infty \text{ (if there is no path from vertex i to j)} \end{cases}$$

where ∞ is assumed to be a very large number.

We will also find a predecessor matrix that will help us construct the shortest path. Any element Pred[i][j] of this predecessor matrix can be defined as-

$$\text{red}[i][j] = \begin{cases} \text{Predecessor of vertex j in shortest path from vertex i to vertex j} \\ \text{NIL (if there is no path from vertex i to j)} \end{cases}$$

We will take a graph with n vertices numbered from 0 to n-1, so we can take NIL to be equal to -1.

In Warshall's algorithm we had worked on adjacency matrix, here we will work on weighted adjacency matrix because to find the shortest paths we need to know the weight(or length) of each edge. In Warshall's algorithm we had found out the matrices P_{-1}, P_0, P_1, ...P_{n-1}, here we will find the matrices D_{-1}, D_0, D_1,.....D_{n-1} where an element $D_k[i][j]$ can be defined as-

$$_k[i][j] = \begin{cases} \text{length of shortest path from vertex i to vertex j,} \\ \text{using only vertices 0,1,2,.....k as intermediate vertices.} \\ \infty \text{ (if there is no path from vertex i to vertex j using vertices 0,1, 2, ...k)} \end{cases}$$

We will also find predecessor matrices $Pred_{-1}$, $Pred_0$,..... $Pred_{n-1}$, where any element $Pred_k[i][j]$ can be defined as-

$$\text{ed}_k[i][j] = \begin{cases} \text{Predecessor of j on the shortest path from vertex i to vertex j,} \\ \text{using only vertices 0,1,2,.....k as intermediate vertices.} \\ -1 \text{ (if there is no path from vertex i to vertex j using vertices 0,1, 2, ...k)} \end{cases}$$

Hence we can say that

$_{-1}[i][j]$ = length of an edge from i to j

$_0[i][j]$ = length of shortest path from i to j using only vertex 0

$_1[i][j]$ = length of shortest path from i to j using only vertices 0, 1

$_2[i][j]$ = length of shortest path from i to j using only vertices 0, 1, 2

.....

$_{n-1}[i][j]$ = length of shortest path from i to j using only vertices 0, 1, 2, ... n-1

We can find matrix D_{-1} from the weighted adjacency matrix by replacing all zero entries by ∞.

$$D_{-1}[i][j] = \begin{cases} \text{length of edge from vertex i to vertex j} \\ \infty \text{ (if there is no edge vertex i to vertex j} \end{cases}$$

The elements of matrix $Pred_{-1}$ can be defined as-

$$Pred_{-1}[i][j] = \begin{cases} i \text{ (if there is an edge from vertex i to vertex j)} \\ -1 \text{ (if there is no edge from vertex i to vertex j} \end{cases}$$

If there are n vertices in the graph then matrix D_{n-1} will represent the shortest path matrix D. Now our purpose is to find out matrices $D_0, D_1, D_2, \ldots\ldots, D_{n-1}$. We have already found out matrix D_{-1} by weighted adjacency matrix. Now if we know how to find out the matrix D_k from matrix D_{k-1} then we can easily find out matrices $D_0, D_1, D_2 \ldots\ldots\ldots D_{n-1}$ also. So let us see how we can find out matrix D_k from matrix D_{k-1}.

We have seen in Warshall's algorithm that $P_k[i][j]=1$ if any of these two conditions is true.

1. $P_{k-1}[i][j]=1$
2. $P_{k-1}[i][k]=1$ and $P_{k-1}[k][j]=1$

This means there can be a path from vertex i to j using vertices 0,1, 2,.....k in two conditions

1. There is a path from vertex i to vertex j using only vertices 0, 1,.....k-1 (path P1)
2. There is a path from vertex i to vertex k using only vertices 0, 1,.....k-1 and there is a path from vertex k to vertex j using only vertices 0,1,.....k-1. (path P2)

Length of first path P1 will be $D_{k-1}[i][j]$
Length of second path P2 will be $D_{k-1}[i][k] + D_{k-1}[k][j]$

Now we will compare the length of these two paths.

(i) If $(D_{k-1}[i][k] + D_{k-1}[k][j]) < D_{k-1}[i][j]$
This means that path from i to j will be shorter if we use k as an intermediate vertex.
Thus $D_k[i][j] = D_{k-1}[i][k] + D_{k-1}[k][j]$
$Pred_k[i][j] = Pred_{k-1}[k][j]$

(ii) If $(D_{k-1}[i][k] + D_{k-1}[k][j]) >= D_{k-1}[i][j]$
This means that path from i to j is not improved if we use k as an intermediate vertex.
Thus $D_k[i][j] = D_{k-1}[i][j]$
$Pred_k[i][j] = Pred_{k-1}[i][j]$

We select the smaller one from the two paths P1 and P2.

Hence value of $D_k[i][j]$ = Minimum($D_{k-1}[i][j]$, $D_{k-1}[i][k] + D_{k-1}[k][j]$)

Now let us take a graph and find out the shortest path matrix for it.

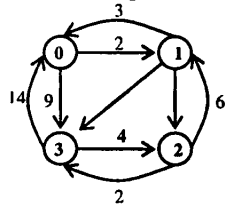


Figure 7.60

Weighted adjacency matrix for this graph is-

$$W = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 2 & 0 & 9 \\ 3 & 0 & 4 & 7 \\ 0 & 6 & 0 & 2 \\ 14 & 0 & 4 & 0 \end{bmatrix} \end{array}$$

We can easily write matrices D_{-1} and $Pred_{-1}$ from matrix W.

$$D_{-1} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \infty & 2 & \infty & 9 \\ 3 & \infty & 4 & 7 \\ \infty & 6 & \infty & 2 \\ 14 & \infty & 4 & \infty \end{bmatrix} \end{array} \qquad Pred_{-1} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} -1 & 0 & -1 & 0 \\ 1 & -1 & 1 & 1 \\ -1 & 2 & -1 & 2 \\ 3 & -1 & 3 & -1 \end{bmatrix} \end{array}$$

ow let us see how we can find matrix D_0 from matrix D_{-1}. If we go through vertex 0 and find a smaller path en we replace the older path with this smaller one. The calculation of some entries of the matrix is shown elow.

Find $D_0[0][0]$
$D_{-1}[0][0] + D_{-1}[0][0] > D_{-1}[0][0]$ => No change
(9999 + 9999 > 9999)
Find $D_0[1][0]$
$D_{-1}[1][0] + D_{-1}[0][0] > D_{-1}[1][0]$ => No change
(3 + 9999 > 3)
Find $D_0[1][1]$
$D_{-1}[1][0] + D_{-1}[0][1] < D_{-1}[1][1]$
(3 + 2 < 9999)
$D_0[1][1] = D_{-1}[1][0] + D_{-1}[0][1] = 5$
$Pred_0[1][1] = Pred_{-1}[0][1] = 0$
Find $D_0[3][1]$
$D_{-1}[3][0] + D_{-1}[0][1] < D_{-1}[3][1]$
(14 + 2 < 9999)
$D_0[3][1] = D_{-1}[3][0] + D_{-1}[0][1] = 16$
$Pred_0[3][1] = Pred_{-1}[0][1] = 0$

he changed values are shown in bold in the matrix.

$$D_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \infty & 2 & \infty & 9 \\ 3 & \mathbf{5} & 4 & 7 \\ \infty & 6 & \infty & 2 \\ 14 & \mathbf{16} & 4 & \mathbf{23} \end{bmatrix} \end{array} \qquad Pred_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} -1 & 0 & -1 & 0 \\ 1 & \mathbf{0} & 1 & 1 \\ -1 & 2 & -1 & 2 \\ 3 & \mathbf{0} & 3 & \mathbf{0} \end{bmatrix} \end{array}$$

ow we have to find the matrices D_1 and $Pred_1$, calculation of some entries are shown.

Find $D_1[1][3]$
$D_0[1][1] + D_0[1][3] > D_0[1][3]$ => No change
(5 + 7 > 7)
Find $D_1[2][0]$
$D_0[2][1] + D_0[1][0] < D_0[2][0]$
(6 + 3 < 9999)
$D_1[2][0] = D_0[2][1] + D_0[1][0] = 9$
$Pred_1[2][0] = Pred_0[1][0] = 1$
Find $D_1[2][2]$
$D_0[2][1] + D_0[1][2] < D_0[2][2]$
(6 + 4 < 9999)
$D_1[2][2] = D_0[2][1] + D_0[1][2] = 10$
$Pred_1[2][2] = Pred_0[1][2] = 1$
Find $D_1[3][0]$
$D_0[3][1] + D_0[1][0] > D_0[3][0]$ => No change
(16 + 3 > 14)

$$D_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \mathbf{5} & 2 & \mathbf{6} & 9 \\ 3 & 5 & 4 & 7 \\ \mathbf{9} & 6 & \mathbf{10} & 2 \\ 14 & 16 & 4 & 23 \end{bmatrix} \end{array} \qquad Pred_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \mathbf{1} & 0 & \mathbf{1} & 0 \\ 1 & 0 & 1 & 1 \\ \mathbf{1} & 2 & \mathbf{1} & 2 \\ 3 & 0 & 3 & 0 \end{bmatrix} \end{array}$$

Similarly we can find matrices D_2, $Pred_2$, D_3 and $Pred_3$.

$$D_2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 5 & 2 & 6 & \underline{8} \\ 3 & 5 & 4 & \underline{6} \\ 9 & 6 & 10 & 2 \\ 13 & 10 & 4 & 6 \end{bmatrix} \end{array} \qquad Pred_2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & \underline{2} \\ 1 & 0 & 1 & \underline{2} \\ 1 & 2 & 1 & 2 \\ \underline{1} & \underline{2} & 3 & \underline{2} \end{bmatrix} \end{array}$$

$$D = D_3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 5 & 2 & 6 & 8 \\ 3 & 5 & 4 & 6 \\ 9 & 6 & \underline{6} & 2 \\ 13 & 10 & 4 & 6 \end{bmatrix} \end{array} \qquad Pred = Pred_3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 2 \\ 1 & 0 & 1 & 2 \\ 1 & 2 & \underline{3} & 2 \\ 1 & 2 & 3 & 2 \end{bmatrix} \end{array}$$

The matrix D_3 is the shortest path matrix D and the matrix $Pred_3$ is the predecessor matrix.

Suppose we have to find the shortest path from vertex 3 to vertex 0. The value of D[3][0] is 13 and it is the length of this shortest path. We can construct the path from matrix Pred.

Pred[3][0] is 1 => predecessor of vertex 0 on shortest path from 3 to 0 is vertex 1
Pred[3][1] is 2 => predecessor of vertex 1 on shortest path from 3 to 1 is vertex 2
Pred[3][2] is 3 => predecessor of vertex 2 on shortest path from 3 to 2 is vertex 3

So the shortest path is 3→2→1→0

If any value D[i][j] is infinity, it means that there is no path from vertex i to vertex j. In the above example we don't have any infinity in the shortest path matrix.

This algorithm can also be used for cycle detection, if there is no cycle in the graph then all diagonal elements will be infinity in the last matrix, otherwise there will be finite values along the diagonal corresponding to vertices which are in the cycle. For example if D[i][i] is a finite value, then it denotes that vertex i is a part of cycle. If this finite value is negative, then it denotes the presence of a negative cycle in the graph and in this case shortest paths are not defined.

```
/*P7.11 Program to find shortest path matrix by Modified Warshall's algorithm*/
#include<stdio.h>
#include<stdlib.h>
#define infinity 9999
#define MAX 100
int n; /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Weighted Adjacency matrix*/
int D[MAX][MAX]; /*Shortest Path Matrix*/
int Pred[MAX][MAX]; /*Predecessor Matrix*/
void create_graph();
void FloydWarshalls();
void findPath(int s,int d);
void display(int matrix[MAX][MAX],int n);
main()
{
        int s,d;
        create_graph();
        FloydWarshalls();
        while(1)
        {
                printf("Enter source vertex(-1 to exit)  : ");
                scanf("%d",&s);
                if(s==-1)
                        break;
                printf("Enter destination vertex : ");
                scanf("%d",&d);
                if(s<0 || s>n-1 || d<0 || d>n-1)
                {
                        printf("Enter valid vertices \n\n");
```

```
                continue;
        }
        printf("Shortest path is : ");
        findPath(s,d);
        printf("Length of this path is %d\n",D[s][d]);
    }
*End of main()*/
id FloydWarshalls()

    int i,j,k;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
        {
            if(adj[i][j]==0)
            {
                D[i][j] = infinity;
                Pred[i][j] = -1;
            }
            else
            {
                D[i][j] = adj[i][j];
                Pred[i][j] = i;
            }
        }
    for(k=0; k<n; k++)
    {
        for(i=0; i<n; i++)
          for(j=0; j<n; j++)
            if(D[i][k]+D[k][j] < D[i][j])
            {
                D[i][j] = D[i][k]+D[k][j];
                Pred[i][j] = Pred[k][j];
            }
    }
    printf("Shortest path matrix is :\n");
    display(D,n);
    printf("\n\nPredecessor matrix is :\n");
    display(Pred,n);
    for(i=0; i<n; i++)
        if(D[i][i]<0)
        {
            printf("Error : negative cycle\n");
            exit(1);
        }
End of FloydWarshalls()*/
d findPath(int s,int d)

    int i,path[MAX],count;
    if(D[s][d]==infinity)
    {
        printf("No path \n");
        return;
    }
    count = -1;
    do
    {
        path[++count] = d;
        d = Pred[s][d];
    }while(d!=s);
    path[++count] = s;
    for(i=count; i>=0; i--)
        printf("%d ",path[i]);
    printf("\n");
nd of findPath()*/
```

```
void display(int matrix[MAX][MAX],int n)
{
        int i,j;
        for(i=0; i<n; i++)
        {
                for(j=0; j<n; j++)
                        printf("%7d",matrix[i][j]);
                printf("\n");
        }
}/*End of display()*/
```

The function `create_graph()` is same as in Program P7.9.

7.14 Minimum spanning tree

The sum of weights of edges of different spanning trees of a graph may be different. A spanning tree of graph G whose sum of weights is minimum amongst all spanning trees of G, is called the Minimum Spanning Tree of G. Let us take a graph and draw its different spanning trees.

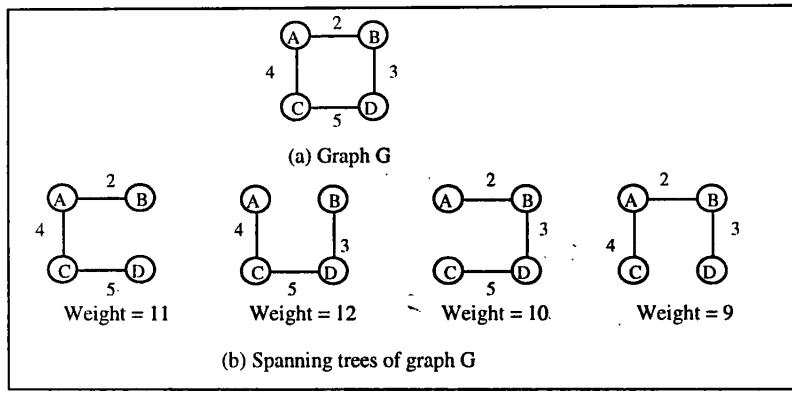


Figure 7.61

Here the tree with weight 9 is the minimum spanning tree. It is not necessary that a graph has unique minimum spanning tree. If there are duplicate weights in the graph then more than one spanning trees are possible, but if the all weights are unique then there will be only one minimum spanning tree.

Minimum spanning tree gives us the most economical way of connecting all the vertices in a graph. For example, in a network of computers we can connect all the computers with the least cost if we construct a minimum spanning tree for the graph where the vertices are computers. Similarly in a telephone communication network we can connect all the cities in the network with the least possible cost.

There are many ways for creating minimum spanning tree but the most famous methods are Prim's and Kruskal's algorithm. Both these methods use the greedy approach.

7.14.1 Prim's Algorithm

In this algorithm we start with an arbitrary vertex as the root and at each step a vertex is added to the tree till all the vertices are in the tree.

The method of making minimum spanning tree from Prim's algorithm is like Dijkstra's algorithm for shortest paths. Each vertex is given a status, which can be permanent or temporary. Initially all the vertices are temporary and at each step of the algorithm, a temporary vertex is made permanent. The process stops when all the vertices are made permanent. Making a vertex permanent means that it has been included in the tree. The temporary vertices are those vertices which have not been added to the tree.

We label each vertex with length and predecessor. The label length represents the weight of the shortest edge connecting the vertex to a permanent vertex and predecessor represents that permanent vertex. Once a vertex is made permanent, it is not relabeled. Only temporary vertices will be relabeled if required.

Applying the greedy approach, the temporary vertex that has the minimum value of length is made permanent. In other words we can say that the temporary vertex which is adjacent to a permanent vertex by an edge of least weight is added to the tree.

The steps for making a minimum spanning tree by Prim's algorithm are as-

A) Initialize the length of all vertices to infinity and predecessors of all vertices to NIL. Make the status of all vertices temporary.

B) Select any arbitrary vertex as the root vertex and make its length label equal to 0.

C) From all the temporary vertices in the graph, find out the vertex that has smallest value of length, make t permanent and now this is our current vertex. (If there are many with the same value of length then anyone can be selected)

D) Examine all the temporary vertices adjacent to the current vertex. Suppose `current` is the current vertex and v is a temporary vertex adjacent to `current`.

(i) If weight (`current`,v) < length(v)
Relabel the vertex v
Now length(v) = weight(`current`,v)
predecessor(v) = `current`

(ii) If weight (`current`,v) >= length(v)
Vertex v is not relabelled

E) Repeat steps (C) and (D) till there are no temporary vertices left, or all the temporary vertices left have length equal to infinity. If the graph is connected, then the procedure will stop when all n vertices have been made permanent and n-1 edges are added to the spanning tree. If the graph is not connected, then those vertices that are not reachable from the root vertex will remain temporary with length infinity. In this case no spanning tree is possible.

Let us take an undirected connected graph and construct the minimum spanning tree

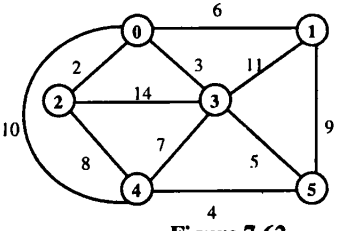
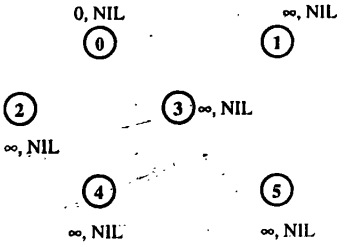


Figure 7.62

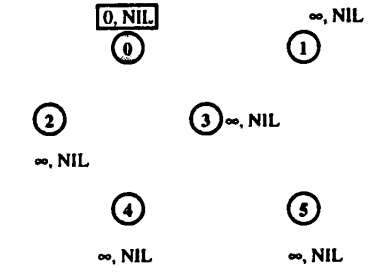
Initially length values for all the vertices are set to a very large number(larger than weight of any edge). Suppose ∞ is such a number. We have taken the predecessor of all vertices NIL(-1) in the beginning.

Initially all the vertices are temporary. We select the vertex 0 as the root vertex and make its length label equal to zero.



Vertex	length	Prede-cessor	Status
0	0	NIL	Temp
1	∞	NIL	Temp
2	∞	NIL	Temp
3	∞	NIL	Temp
4	∞	NIL	Temp
5	∞	NIL	Temp

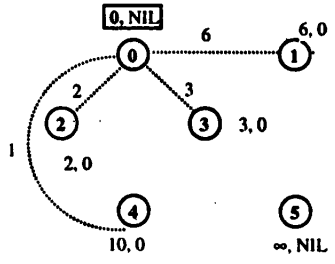
From all the temporary vertices, vertex 0 has the smallest length, so it will be made permanent. This is the first vertex to be included in the tree. Its predecessor will remain NIL. Now vertex 0 is the current vertex.



Vertex	length	Prede-	Status
0	0	NIL	Perm
1	∞	NIL	Temp
2	∞	NIL	Temp
3	∞	NIL	Temp
4	∞	NIL	Temp
5	∞	NIL	Temp

Vertices 1, 2, 3, 4 are temporary vertices adjacent to 0

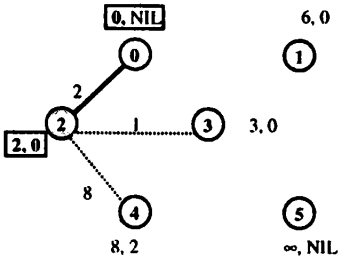
weight(0,1) < length(1) 6 < ∞ Relabel 1
predecessor[1]=0 , length[1]=6
weight(0,2) < length(2) 2 < ∞ Relabel 2
predecessor[2]=0 , length[2]=2
weight(0,3) < length(3) 3 < ∞ Relabel 3
predecessor[3]=0 , length[3]=3
weight(0,4) < length(4) 10 < ∞ Relabel 4
predecessor[4]= 0, length[4]=10



Vertex	length	Prede-cessor	Status
0	0	NIL	Perm
1	6	0	Temp
2	2	0	Temp
3	3	0	Temp
4	10	0	Temp
5	∞	NIL	Temp

From all the temporary vertices, vertex 2 has the smallest length so make it permanent i.e. include it in the tree. Its predecessor is 0, so the edge that is added to the tree is (0,2). Now vertex 2 is the current vertex. Vertices 3, 4 are temporary vertices adjacent to vertex 2.

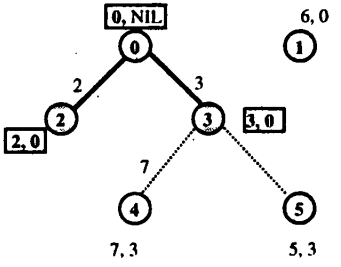
weight(2,3) > length(3) 14 > 3 Don't Relabel 3
weight(2,4) < length(4) 8 < 10 Relabel 4
predecessor[4]=2 , length[4]=8



Vertex	length	Prede-	Status
0	0	NIL	Perm
1	6	0	Temp
2	2	0	Perm
3	3	0	Temp
4	8	2	Temp
5	∞	NIL	Temp

From all temporary vertices, vertex 3 has the smallest value of length so make it permanent. Its predecessor is 0, so the edge (0,3) is included in the tree. Now vertex 3 is the current working vertex. Vertices 1, 4, 5 are temporary vertices adjacent to vertex 3

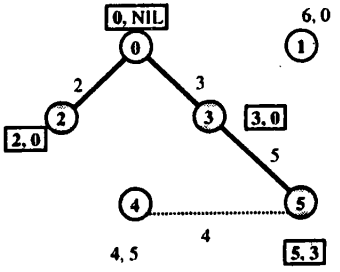
weight(3,1) > length(1) 1 1>6 Don't relabel 1
weight(3,4) < length(4) 7 < 8 Relabel 4
predecessor[4]=3 , length[4]=7
weight(3,5) < length(5) 5 < ∞ Relabel 5
predecessor[5]=3 , length[5]=5



Vertex	length	Prede-cessor	Status
0	0	NIL	Perm
1	6	0	Temp
2	2	0	Perm
3	3	0	Perm
4	7	3	Temp
5	5	3	Temp

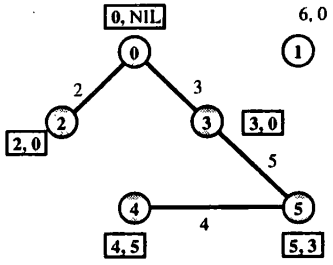
From all temporary vertices, vertex 5 has the smallest length so make it permanent. Its predecessor is 3 so include edge (3,5) in the tree. Now vertex 5 is the current vertex. Vertices 1, 4 are temporary vertices adjacent to vertex 5

weight(5,1) > length(1) 9 > 6 Don't relabel 1
weight(5,4) < length(4) 4 < 7 Relabel 4
predecessor[4]=5 , length[4]=4



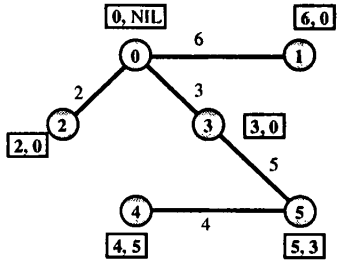
Vertex	length	Prede-cessor	Status
0	0	NIL	Perm
1	6	0	Temp
2	2	0	Perm
3	3	0	Perm
4	4	5	Temp
5	5	3	Perm

From all temporary vertices, vertex 4 has the smallest length so make it permanent. Its predecessor is 5, so include edge (5,4) in the tree. Now vertex 4 is the current vertex. There are no temporary vertices adjacent to 4



Vertex	length	Prede-cessor	Status
0	0	NIL	Perm
1	6	0	Temp
2	2	0	Perm
3	3	0	Perm
4	4	5	Perm
5	5	3	Perm

Vertex 1 is the only temporary vertex left and its length is 6, so make it permanent. Its predecessor is 0, so edge (0,1) is included in the tree.



Now all the vertices are permanent so we stop our procedure.

Vertex	length	Prede-cessor	Status
0	0	NIL	Perm
1	6	0	Perm
2	2	0	Perm
3	3	0	Perm
4	4	5	Perm
5	5	3	Perm

Now we have a complete minimum spanning tree. The edges that belong to minimum spanning tree are - (0,1), (0,2), (0,3), (5,4), (3,5)

Weight of minimum spanning tree will be-

$6 + 2 + 3 + 4 + 5 = 20$

Now let us take a graph that is not connected.

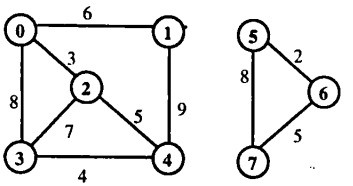


Figure 7.63

fter making vertices 0,2,4,3,1 permanent, the situation would be-

Vertex	length	Prede-cessor	Status
0	0	NIL	Perm
1	6	0	Perm
2	3	0	Perm
3	4	4	Perm
4	5	2	Perm
5	∞	NIL	Temp
6	∞	NIL	Temp
7	∞	NIL	Temp

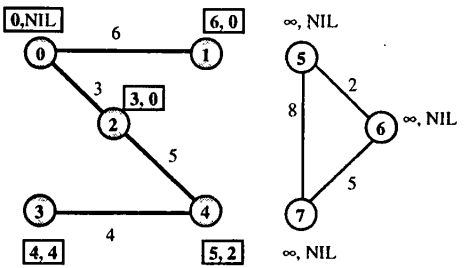


Figure 7.64

ertices 5, 6, 7 are temporary with length infinity so we stop the procedure and state that the graph is not nnected and hence no spanning tree is possible.

```
P7.12 Program for creating minimum spanning tree using Prim's algorithm*/
nclude<stdio.h>
nclude<stdlib.h>
efine MAX 10
efine TEMP 0
efine PERM 1
efine infinity 9999
efine NIL -1
ruct edge

      int u;
      int v;

t n;
t adj[MAX][MAX];
t predecessor[MAX];
t status[MAX];
t length[MAX];
id create_graph();
id maketree(int r, struct edge tree[MAX]);
t min_temp();
in()

      int wt_tree = 0;
      int i, root;
      struct edge tree[MAX];
      create_graph();
      printf("Enter root vertex : ");
      scanf("%d",&root);
      maketree(root, tree);
      printf("Edges to be included in spanning tree are : \n");
      for(i=1; i<=n-1; i++)
      {
            printf("%d->",tree[i].u);
            printf("%d\n",tree[i].v);
            wt_tree += adj[tree[i].u][tree[i].v];
```

```
        }
        printf("Weight of spanning tree is : %d\n", wt_tree);
}/*End of main()*/
void maketree(int r, struct edge tree[MAX])
{
        int current,i;
        int count = 0;/*number of vertices in the tree*/
        for(i=0; i<n; i++)      /*Initialize all vertices*/
        {
                predecessor[i] = NIL;
                length[i] = infinity;
                status[i] = TEMP;
        }
        length[r] = 0; /*Make length of root vertex 0*/
        while(1)
        {
                /*Search for temporary vertex with minimum length
                and  make it current vertex*/
                current = min_temp();
                if(current==NIL)
                {
                  if(count==n-1) /*No temporary vertex left*/
                      return;
                  else /*Temporary vertices left with length infinity*/
                  {
                    printf("Graph is not connected, No spanning tree possible\n");
                    exit(1);
                  }
                }
                status[current] = PERM; /*Make the current vertex permanent*/
                /*Insert the edge (predecessor[current], current) into the tree,
                except when the current vertex is root*/
                if(current!=r)
                {
                   count++;
                   tree[count].u = predecessor[current];
                   tree[count].v = current;
                }
                for(i=0; i<n; i++)
                        if(adj[current][i]>0 && status[i]==TEMP)
                                if(adj[current][i] < length[i])
                                {
                                        predecessor[i] = current;
                                        length[i] = adj[current][i];
                                }
        }
}/*End of make_tree()*/
/*Returns the temporary vertex with minimum value of length, Returns NIL if no temporary
vertex left or all temporary vertices left have pathLength infinity*/
int min_temp()
{
        int i;
        int min = infinity;
        int k = -1;
        for(i=0; i<n; i++)
        {
                if(status[i]==TEMP && length[i]<min)
                {
                        min = length[i];
                        k = i;
                }
        }
        return k;
}/*End of min_temp()*/
```

```
id create_graph()

    int i,max_edges,origin,destin,wt;
    printf("Enter number of vertices : ");
    scanf("%d",&n);
    max_edges = n*(n-1)/2; /*Undirected graph*/
    for(i=1; i<=max_edges; i++)
    {
            printf("Enter edge %d(-1 -1 to quit) : ",i);
            scanf("%d %d",&origin,&destin);
            if((origin==-1) && (destin==-1))
                    break;
            printf("Enter weight for this edge : ");
            scanf("%d",&wt);
            if(origin>=n || destin>=n || origin<0 || destin<0)
            {
                    printf("Invalid edge!\n");
                    i--;
            }
            else
            {
                    adj[origin][destin] = wt;
                    adj[destin][origin] = wt;
            }
    }
End of create_graph()*/
```

14.2 Kruskal's Algorithm

s approach of constructing a minimum spanning tree was formulated by J.B.Kruskal and is named after him Kruskal's algorithm.

In this algorithm, initially we take a forest of n distinct trees for all n vertices of the graph. So at the start of orithm, each tree is a single vertex tree and no edges are there. In each step of the algorithm an edge is mined and it is included in the spanning tree if its inclusion does not form a cycle. If this edge forms a cycle n it is rejected.

Now the question arises as to which edge should be considered for examining. Here we apply the greedy roach and use the edge with the minimum weight. For this we can maintain a list of edges sorted in ending order of their weights. At each step we take an edge from this list and include it if it does not form a le. If the edge forms a cycle, we reject it. The algorithm completes when n-1 edges have been included. If graph contains less than n-1 edges, then it means that graph is not connected and no spanning tree is sible.

nitially we have a forest of n trees, whenever an edge is included two distinct trees are joined into a single . So at any stage of this algorithm we don't have a single tree as in Prim's, but we have a forest of trees. enever an edge is inserted, the number of trees in the forest decreases by one, and at the end when n-1 edges e been included we are left with only one tree which is our minimum spanning tree.Let us take a graph and ute a minimum spanning tree by Kruskal's algorithm.

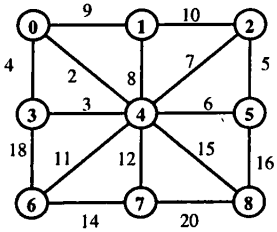


Figure 7.65

Initially we take a forest of 9 trees, with each tree consisting of a single vertex. All the edges are examined increasing order of their weight.

Edge 0-4, wt = 2	Inserted, see figure 7.66(b)
Edge 3-4, wt = 3	Inserted, see figure 7.66(c)
Edge 0-3, wt = 4	Not inserted, forms cycle 0-3-4-0 in figure 7.66 (c)
Edge 2-5, wt = 5	Inserted, see figure 7.66(d)
Edge 4-5, wt = 6	Inserted, see figure 7.66(e)
Edge 2-4, wt = 7	Not inserted, forms cycle 2-4-5-2 in figure 7.66(e)
Edge 1-4, wt = 8	Inserted, see figure 7.66(f)
Edge 0-1, wt = 9	Not inserted, forms cycle 0-1-4-0 in figure 7.66(f)
Edge 1-2, wt = 10	Not inserted, forms cycle 1-2-5-4-1 in figure 7.66(f)
Edge 4-6, wt = 11	Inserted, see figure 7.66(g)
Edge 4-7, wt = 12	Inserted, see figure 7.66(h)
Edge 6-7, wt = 14	Not inserted, forms cycle 4-6-7-4 in figure 7.66(h)
Edge 4-8, wt = 15	Inserted, see figure 7.66(i)

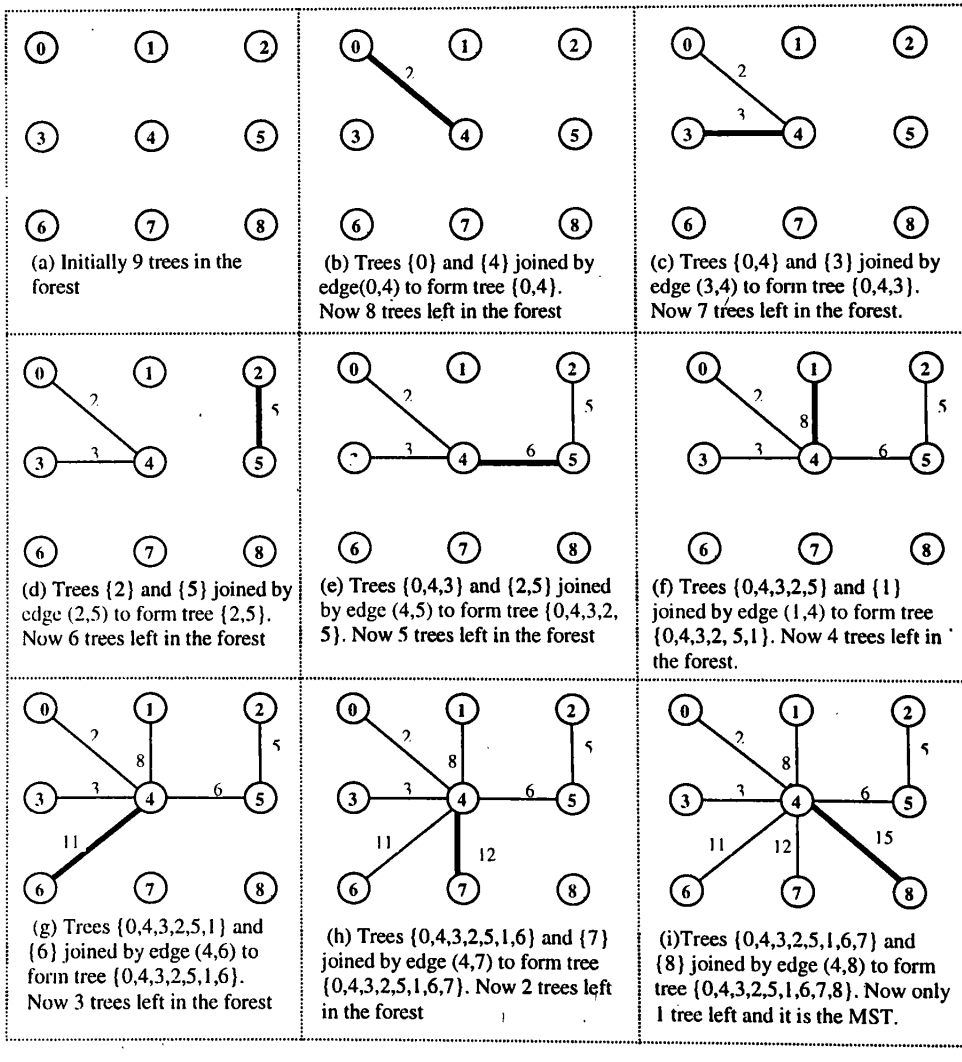

(a) Initially 9 trees in the forest
(b) Trees {0} and {4} joined by edge(0,4) to form tree {0,4}. Now 8 trees left in the forest
(c) Trees {0,4} and {3} joined by edge (3,4) to form tree {0,4,3}. Now 7 trees left in the forest.
(d) Trees {2} and {5} joined by edge (2,5) to form tree {2,5}. Now 6 trees left in the forest
(e) Trees {0,4,3} and {2,5} joined by edge (4,5) to form tree {0,4,3,2, 5}. Now 5 trees left in the forest
(f) Trees {0,4,3,2,5} and {1} joined by edge (1,4) to form tree {0,4,3,2, 5,1}. Now 4 trees left in the forest.
(g) Trees {0,4,3,2,5,1} and {6} joined by edge (4,6) to form tree {0,4,3,2,5,1,6}. Now 3 trees left in the forest
(h) Trees {0,4,3,2,5,1,6} and {7} joined by edge (4,7) to form tree {0,4,3,2,5,1,6,7}. Now 2 trees left in the forest
(i)Trees {0,4,3,2,5,1,6,7} and {8} joined by edge (4,8) to form tree {0,4,3,2,5,1,6,7,8}. Now only 1 tree left and it is the MST.

Figure 7.66

The resulting minimum spanning tree is-

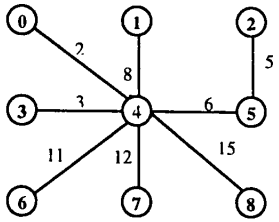


Figure 7.67

Let us see how we can implement this algorithm. We examine all the edges one by one starting from the llest edge. To decide whether the selected edge should be included in the spanning tree or not, we will mine the two vertices connecting the edge. If the two vertices belong to the same tree, it means that they are ady connected and adding this edge would result in a cycle. So we will insert an edge in the spanning tree if its vertices are in different trees.

Now the question is how to decide whether two vertices are in the same tree or not. We will keep record of father of every vertex. Since this is a tree, every vertex will have only one distinct father. We will recognize n tree by a root vertex and a vertex will be a root if its father is NIL(-1). Initially we have only single vertex s; each vertex is a root vertex so we will take father of all vertices as NIL. For finding out, to which tree a ex belongs, we will find out the root of that tree. So we will traverse all the ancestors of the vertex till we h a vertex whose father is NIL. This will be the root of the tree to which the vertex belongs.

Now we know the root of both vertices of an edge, if roots are same means both vertices are in the same tree are already connected so this edge is rejected. If the roots are different, then we will insert this edge into the nning tree and we will join the two trees, which are having these two vertices. For joining the two trees, we make root of one tree as the father of root-of another tree.

After joining two trees, all the vertices of both trees will be connected and have the same root.Initially father very vertex is NIL(-1), and hence every vertex is a root vertex.

ex	0	1	2	3	4	5	6	7	8
er	N	N	N	N	N	N	N	N	N

ge	Wt	vl	v2	root of vl	root of v2	Result									
	2	0	4	0	4	Inserted father[4]=0	vertex	**0**	1	2	3	**4**	5	6	7	8
							father	N	N	N	N	**0**	N	N	N	N
	3	3	4	3	0	Inserted father[0]=3	vertex	**0**	1	2	3	4	5	6	7	8
							father	**3**	N	N	N	0	N	N	N	N
	4	0	3	3	3	Not inserted	vertex	0	1	2	3	4	5	6	7	8
							father	3	N	N	N	0	N	N	N	N
	5	2	5	2	5	Inserted father[5]=2	vertex	0	1	2	3	4	**5**	6	7	8
							father	3	N	N	N	0	**2**	N	N	N
	6	4	5	3	2	Inserted father[2]=3	vertex	0	1	**2**	3	4	5	6	7	8
							father	3	N	**3**	N	0	2	N	N	N
	7	2	4	3	3	Not inserted	vertex	0	1	2	3	4	5	6	7	8
							father	3	N	3	N	0	2	N	N	N
	8	1	4	1	3	Inserted father[3]=1	vertex	0	1	2	**3**	4	5	6	7	8
							father	3	**N**	3	**1**	0	2	N	N	N
	9	0	1	1	1	Not inserted	vertex	0	1	2	3	4	5	6	7	8
							father	3	**N**	3	1	0	2	N	N	N
	10	1	2	1	1	Not inserted	vertex	0	**1**	2	3	4	5	6	7	8
							father	3	N	3	1	0	2	N	N	N
	11	4	6	1	6	Inserted father[6]=1	vertex	0	1	2	3	4	5	**6**	7	8
							father	3	N	3	1	0	2	**1**	N	N
	12	4	7	1	7	Inserted father[7]=1	vertex	0	1	2	3	4	5	6	**7**	8
							father	3	N	3	1	0	2	1	**1**	N
	14	6	7	1	1	Not inserted	vertex	0	1	2	3	4	5	6	7	8
							father	3	N	3	**1**	0	2	1	1	N
	15	4	8	1	8	Inserted father[8]=1	vertex	0	1	2	3	4	5	6	7	**8**
							father	3	N	3	1	0	2	1	1	**1**

The minimum spanning tree should contain n-1 edges where n is the number of vertices in the graph. This graph contains 9 vertices so after including 8 edges in the spanning tree, we will not examine other edges of the graph and stop our process.

Edges included in this spanning tree are (0,4), (3,4), (2,5), (4,5), (1,4), (4,6), (4,7), (4,8)
Weight of this spanning tree is 2 + 3 + 5 + 6 + 8 + 11 + 12 +15 = 62

We will take an array named father and if the index of the array is considered as the vertex, then the element present at that index will represent the father of that vertex.

To obtain the edges in ascending order we can insert them in a priority queue. We will take a linked priority queue of edges in increasing order of their weights.

In Prims algorithm we have a single tree at all the stages of the algorithm, while in Kruskal's algorithm we have a tree only in the end. Kruskal's algorithm is faster than Prim's because in latter we may have to consider an edge several times, but in the former an edge is considered only once.

```
/* P7.13 Program for creating a minimum spanning tree by Kruskal's algorithm*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define NIL -1
struct edge
{
    int u;
    int v;
    int weight;
    struct edge *link;
}*front = NULL;
void make_tree(struct edge tree[]);
void insert_pque(int i,int j,int wt);
struct edge *del_pque();
int isEmpty_pque();
void create_graph();
int n;  /*Total number of vertices in the graph*/
main()
{
        int i;
        struct edge tree[MAX]; /*Will contain the edges of spanning tree*/
        int wt_tree = 0; /*Weight of the spanning tree*/
        create_graph();
        make_tree(tree);
        printf("Edges to be included in minimum spanning tree are :\n");
        for(i=1; i<=n-1; i++)
        {
                printf("%d->",tree[i].u);
                printf("%d\n",tree[i].v);
                wt_tree += tree[i].weight;
        }
        printf("Weight of this minimum spanning tree is : %d\n",wt_tree);
}/*End of main()*/
void make_tree(struct edge tree[])
{
        struct edge *tmp;
        int v1,v2,root_v1,root_v2;
        int father[MAX]; /*Holds father of each vertex*/
        int i,count = 0;    /*Denotes number of edges included in the tree*/
        for(i=0; i<n; i++)
                father[i] = NIL;
        /*Loop till queue becomes empty or till n-1 edges have been inserted in the tree*/
        while(!isEmpty_pque() && count<n-1)
        {
                tmp = del_pque();
                v1 = tmp->u;
```

```
            v2 = tmp->v;
            while(v1!=NIL)
            {
                    root_v1 = v1;
                    v1 = father[v1];
            }
            while(v2!=NIL)
            {
                    root_v2 = v2;
                    v2 = father[v2];
            }
            if(root_v1!=root_v2)/*Insert the edge (v1, v2)*/
            {
                   count++;
                    tree[count].u = tmp->u;
                    tree[count].v = tmp->v;
                    tree[count].weight = tmp->weight;
                    father[root_v2]=root_v1;
            }
      }
      if(count<n-1)
      {
            printf("Graph is not connected, no spanning tree possible\n");
            exit(1);
      }
*End of make_tree()*/
Inserting edges in the linked priority queue*/
id insert_pque(int i,int j,int wt)

      struct edge *tmp,*q;
      tmp = (struct edge *)malloc(sizeof(struct edge));
      tmp->u = i;
      tmp->v = j;
      tmp->weight = wt;
      /*Queue is empty or edge to be added has weight less than first edge*/
      if(front==NULL || tmp->weight<front->weight)
      {
            tmp->link = front;
            front = tmp;
      }
      else
      {
            q = front;
            while(q->link!=NULL && q->link->weight<=tmp->weight)
                  q = q->link;
            tmp->link = q->link;
            q->link = tmp;
            if(q->link==NULL)      /*Edge to be added at the end*/
                  tmp->link = NULL;
      }
*End of insert_pque()*/
Deleting an edge from the linked priority queue*/
ruct edge *del_pque()

      struct edge *tmp;
      tmp = front;
      front = front->link;
      return tmp;
*End of del_pque()*/
 isEmpty_pque()

      if(front==NULL)
            return 1;
```

```
        else
                return 0;
}/*End of isEmpty_pque()*/
void create_graph()
{
        int i,wt,max_edges,origin,destin;

        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edges = n*(n-1)/2;
        for(i=1; i<=max_edges; i++)
        {
                printf("Enter edge %d(-1 -1 to quit): ",i);
                scanf("%d %d",&origin,&destin);
                if((origin==-1) && (destin==-1))
                        break;
                printf("Enter weight for this edge : ");
                scanf("%d",&wt);
                if(origin>=n || destin>=n || origin<0 || destin<0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                        insert_pque(origin,destin,wt);
        }
}/*End of create_graph()*/
```

7.15 Topological Sorting

Topological sorting of a directed acyclic graph is the linear ordering of all the vertices such that if there is a path from vertex u to vertex v, then u comes before v in the ordering. The sequence of vertices in the linear ordering is known as topological order or topological sequence. An example of a graph and its linear ordering is given in figure 7.68.

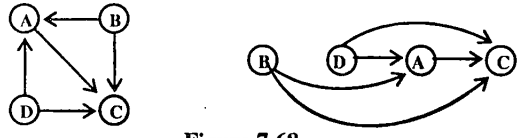


Figure 7.68

We can see that the process of topological sorting linearizes the graph, i.e. we can write all the vertices in a horizontal line such that all the directed edges go from left to right only. This phenomenon is entirely different from the usual sorting techniques.

Topological sorting is possible only in acyclic graphs, i.e. if the graph contains a cycle then no topological order is possible. This is because for any two vertices u and v in the cycle, u precedes v and v precedes u.

There may be more than one topological sequence for a given directed acyclic graph. For example another topological ordering for the graph of figure 7.68 is -

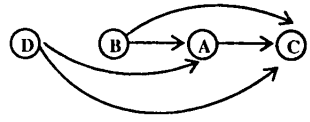


Figure 7.69

Before studying the algorithm for topological sorting, let us first see where it can be used. There are many applications where execution of one task is necessary before starting another task. For example understanding of 'C' language and programming is necessary before starting 'Data Structure through C'. Similarly in this book also we can go to heap sort or binary tree sort only after understanding tree.

To model these types of problems where tasks depend on one another we can draw a directed graph in which ertices represent tasks, and if task x has to be completed before task y then there is a directed edge from x to y.

Suppose a student needs to pass some courses to acquire a degree. The curriculum includes 7 courses named A, B, C, D, E, F, G. Some courses have to be taken before others, for example course paper B can be studied nly after studying papers A, C, D. The prerequisite courses for each course are given in the table below.

Course	Prerequisite courses
A	-
B	A, C, D
C	-
D	C
E	B, D, G
F	A, B, E, G
G	-

Now we have to find the sequence in which the student should take the courses such that before taking up any ourse, all its prerequisite courses are completed. We can represent the information given in the table in the orm of a directed graph. If paper u is a prerequisite for paper v, then there is an edge directed from u to v.

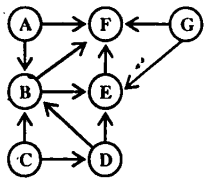


Figure 7.70

Now from this directed graph we can find out the topological order which is the required sequence in which udents should take the courses. We have seen earlier that there may be more than one topological sequence ossible; one of such sequences for the above graph is A-C-G-D-B-E-F.

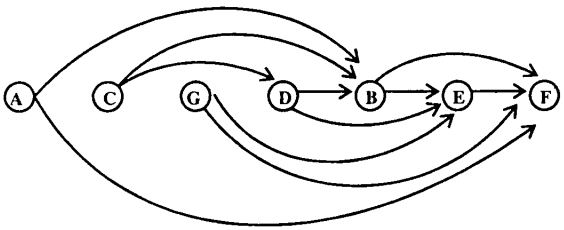


Figure 7.71

Now we will study the algorithm for topological sorting. It finds out the solution using the greedy approach. he procedure is -

Select a vertex with no predecessors (vertex with zero indegree).
Delete this vertex and all edges going out from it.
Repeat this procedure till all the vertices are deleted from the graph.

If in the middle of this procedure we arrive at a situation when no vertex can be deleted, i.e. there is no ertex left with zero indegree; all the vertices in graph have predecessors, then it means that the graph has a ycle. In this case no solution is possible.

Now let us see how we can implement this algorithm. To keep track of all the vertices with zero indegree we an use either a stack or queue. Here we will use a queue and it will temporarily store the vertices with zero degree. We will take a one-d array `topo_order` which will be used to represent the topological order of the ertices. The vertices will be stored in this array in the sequence of their deletion from the queue.

Initially the indegree of all the vertices are computed and the vertices that have zero indegree are inserted into the initially empty queue. A vertex from the queue is deleted and listed in the `topo_order` array. The edges going from this vertex are deleted and the indegrees of its successors are decremented by 1. A vertex is inserted into the queue as soon as its indegree becomes 0. This process continues till all the vertices in the graph are deleted.

Let us take a graph and apply the topological sorting-

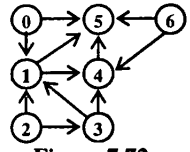


Figure 7.72

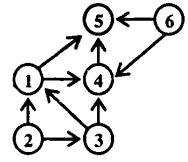
Indegree of vertices are-

In(0)=0, In(1)=3, In(2)=0, In(3)=1, In(4)=3, In(5)=4,In(6)=0

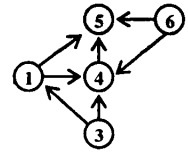
The vertices with zero indegree are 0, 2 and 6. They can be inserted in the queue in any order, this is why more than one topological orders are possible.

Queue : 0, 2, 6

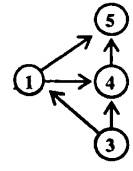
Step 1 - Delete the vertex 0 and edges going from vertex 0.
Queue : 2, 6 topo_order : 0
Updated indegree of vertices: In(1)=2, In(3)=1, In(4)=3, In(5)=3



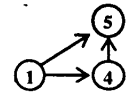
Step 2 - Delete the vertex 2 and edges going from vertex 2
Queue : 6 topo_order : 0, 2
Updated indegree of vertices : In(1)=1, In(3)=0, In(4)=3, In(5)=3
Insert vertex 3 into the queue
Queue : 6, 3



Step 3 - Delete the vertex 6 and edges going from vertex 6.
Queue : 3 topo_order : 0, 2, 6
Updated indegree of vertices: In(1)=1, In(4)=2, In(5)=2



Step 4 - Delete the vertex 3 and edges going from vertex 3.
Queue: Empty topo_order : 0, 2, 6, 3
Updated indegree of vertices: In(1)=0, In(4)=1, In(5)=2
Insert vertex 1 into the queue
Queue: 1



Step 5 - Delete the vertex 1 and edges going from vertex 1.
Delete the vertex 1 and edges going from vertex 1.
Queue: Empty topo_order : 0, 2, 6, 3, 1
Updated indegree of vertices: In(4) = 0, In(5) = 1
Insert vertex 4 into queue
Queue: 4

Step 6 - Delete the vertex 4 and edges going from vertex 4 (5)
Queue: Empty topo_order : 0, 2, 6, 3, 1, 4
Updated indegree of vertices: In(5) = 0
Insert vertex 5 into the queue
Queue: 5

Step 7- Delete the vertex 5 and edges going from vertex 5
Queue: Empty topo_order : 0, 2, 6, 3, 1, 4, 5

Now we have no more vertices in the graph. So the topological sorting of graph will be-
0, 2, 6, 3, 1, 4, 5

If there are vertices remaining in the graph and the queue becomes empty at the end of any step, it implies that the graph contains cycle so we will stop our procedure. In the program we will take an array `indeg`, which will store the indegree of vertices.

```
/*P7.14 Program for topological sorting*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
int n;    /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Adjacency Matrix*/
void create_graph();
int queue[MAX],front = -1,rear = -1;
void insert_queue(int v);
int delete_queue();
int isEmpty_queue();
int indegree(int v);

main()
{
        int i,v,count,topo_order[MAX],indeg[MAX];
        create_graph();
        /*Find the indegree of each vertex*/
        for(i=0; i<n; i++)
        {
                indeg[i] = indegree(i);
                if(indeg[i]==0)
                        insert_queue(i);
        }
        count = 0;
        while(!isEmpty_queue() && count<n)
        {
                v = delete_queue();
                topo_order[++count] = v; /*Add vertex v to topo_order array*/
                /*Delete all edges going fron vertex v */
                for(i=0; i<n; i++)
                {
                        if(adj[v][i]==1)
                        {
                                adj[v][i] = 0;
                                indeg[i] = indeg[i]-1;
                                if(indeg[i]==0)
                                        insert_queue(i);
                        }
                }
        }
        if(count<n)
        {
          printf("No topological ordering possible, graph contains cycle\n");
          exit(1);
        }
        printf("Vertices in topological order are :\n");
        for(i=1; i<=count; i++)
```

```
                printf("%d ",topo_order[i]);
        printf("\n");
}/*End of main()*/
void insert_queue(int vertex)
{
        if(rear==MAX-1)
                printf("Queue Overflow\n");
        else
        {
                if(front==-1)  /*If queue is initially empty */
                        front = 0;
                rear = rear+1;
                queue[rear] = vertex ;
        }
}/*End of insert_queue()*/
int isEmpty_queue()
{
        if(front==-1 || front>rear)
                return 1;
        else
                return 0;
}/*End of isEmpty_queue()*/
int delete_queue()
{
        int del_item;
        if(front==-1 || front>rear)
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        else
        {
                del_item = queue[front];
                front = front+1;
                return del_item;
        }
}/*End of delete_queue() */
int indegree(int v)
{
        int i,in_deg = 0;
        for(i=0; i<n; i++)
                if(adj[i][v]==1)
                        in_deg++;
        return in_deg;
}/*End of indegree()*/
```

The function `create_graph()` is same as in program P7.9.

Exercise

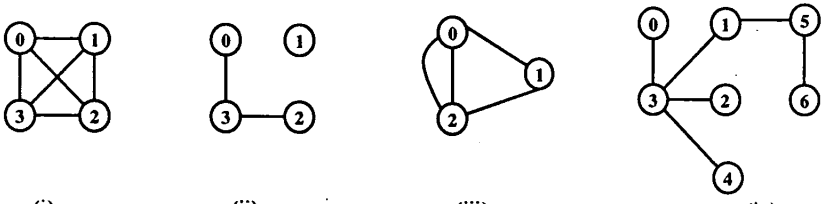
1. Draw a graph corresponding to the following adjacency matrix.

	0	1	2	3	4
0	0	1	0	1	0
1	0	0	0	0	0
2	0	1	0	1	1
3	1	1	0	0	0
4	0	0	0	0	0

(i) Find indegree and outdegree of each vertex.

(ii) Find the path matrix for this graph.
(iii) Is the graph strongly connected.
(iv) Is the graph acyclic.

2. For the following undirected graphs, find the sum of degrees of all the vertices.

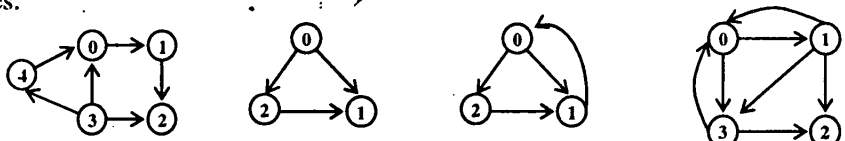


(i) (ii) (iii) (iv)

What is the relation between the number of edges and the sum of degrees of all the vertices.

3. In a graph, the vertices having odd degree are called odd vertices and the vertices having even degree are called even vertices. Prove that number of odd vertices in a graph is even.

4. For the following directed graphs find the sum of indegrees of all vertices and the sum of outdegrees of all the vertices.

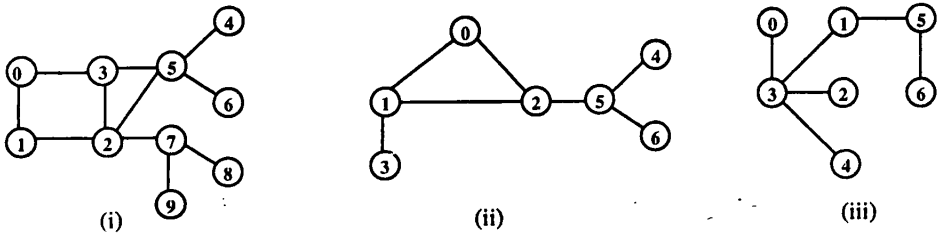


(i) (ii) (iii) (iv)

What is the relation between the number of edges, sum of indegrees and sum of outdegrees of all the vertices.

5. How many edges are there in a regular graph of n vertices having degree d.
(i) A regular graph of degree 2 has 5 vertices. How many edges are there in the graph.
(ii) A regular graph of degree 3 has 4 vertices. How many edges are there in the graph.
(iii) A regular graph of degree 3 has 5 vertices. How many edges are there in the graph.

6. Find the articulation points in the following graphs.

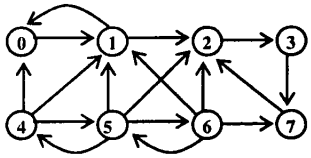


(i) (ii) (iii)

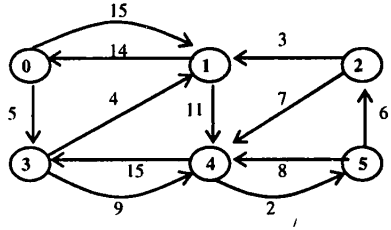
7. Find whether the following graphs are strongly connected or not.

(i) (ii)

8. Find the strongly connected components in the following graph.



9. For the following graph compute the shortest paths from vertex 0 to all other vertices using Dijkstras algorithm.



10. Consider a complete graph having vertices 0, 1, 2, ... n-1.
(i) Construct a minimum spanning tree if weight on any edge (u,v) is u+v
(ii) Construct a minimum spanning tree if weight on any edge (u,v) is | u-v |
(ii) Construct a minimum spanning tree if weight on any edge (u,v) is 5 | u-v |

Sorting

Sorting means arranging the data according to their values in some specified order, where order can be either ascending or descending. For example if we have a list of numbers {6, 2, 8, 1, 4}, then after sorting them in ascending order we get {1, 2, 4, 6, 8} and after sorting them in descending order we get {8, 6, 4, 2, 1}. Here the data that we sorted consists only of numbers, but it may be anything like strings or records. Generally we have to sort a list of records where each record contains several information fields. Sorting is done with respect to a key where key is a part of the record. Sorting these records means rearranging the records so that the key values are in order. The key on which sorting is performed is also known as the sort key.

Suppose we have several records of employees where each record contains three fields viz. name, age and salary. We can sort the records taking any one of these fields as the sort key. The table below shows the unsorted list of records and the sorted lists having name, age and salary as the sort keys one by one.

Name	Age	Salary
Sumit	25	4500
Amit	37	7800
Deepak	45	6000
Neelam	29	3200
Priya	23	9000
Shaman	50	6500
Kiran	39	5500
Chetna	34	8000

Unsorted List L

Name	Age	Salary
Amit	37	7800
Chetna	34	8000
Deepak	45	6000
Kiran	39	5500
Neelam	29	3200
Priya	23	9000
Shaman	50	6500
Sumit	25	4500

List L sorted by name in ascending order

Name	Age	Salary
Priya	23	9000
Sumit	25	4500
Neelam	29	3200
Chetna	34	8000
Amit	37	7800
Kiran	39	5500
Deepak	45	6000
Shaman	50	6500

List L sorted by age in ascending order

Name	Age	Salary
Priya	23	9000
Chetna	34	8000
Amit	37	7800
Shaman	50	6500
Deepak	45	6000
Kiran	39	5500
Sumit	25	4500
Neelam	29	3200

List L sorted by salary in descending order

Figure 8.1 Sorting on different keys

We can see that sorting the data according to different keys arranges the data in different orders.

In our algorithms we will perform sorting on a list of integer values only, so that we can focus on the logic of the algorithm. The extension of these algorithms to sort a list of records is simple. After the discussion of all sort methods, there are two programs in which we can see how to sort records.

Now let us see what is the requirement of sorting and why is it important to keep our data in sorted order. In our daily life, we can see many places where data is kept in sorted order like dictionary, telephone directory, index of books, bank accounts, merit list, roll numbers etc. Imagine the time taken to search for a word in a dictionary if the words were not arranged alphabetically or consider the case when you have to search for a name in telephone directory and the names are not sorted. Suppose you want to know where the topic "Queue" is given in this book, then obviously you will go to the index of this book to find the page number; you directly go to the words starting with 'Q' and in an instant you find your word. This was possible because words in the index were sorted. If the index was not sorted then you had only one option for searching a particular word i.e. one by one. So we see that it is easier and faster to search for an item in data that is sorted. Similarly in computer applications, sorting helps in faster information retrieval and hence the data processing operations become more efficient if data is arranged in some specific order. So practically there is no data processing application that does not perform sorting.

8.1 Sort Order

We can sort the data either in ascending (increasing) or in descending (decreasing) order. If the order is not mentioned then it is assumed to be ascending order. In this chapter we will use ascending order in our examples and algorithms. By making simple modifications, these algorithms can be made to work for descending order also.

In the case of numbers the ascending or descending order is clear. The alphabetic and nonalphabetic characters are generally ordered according to their ASCII values. The strings can be ordered using `strcmp()` function.

8.2 Types of Sorting

There are two types of sorting, internal sorting and external sorting. If the data to be sorted is small enough to be placed in main memory at a time, then the sorting process can take place in main memory and this sorting is called internal sorting. So in internal sorting all the data to be sorted is kept in the main memory during the sorting process.

If there is large amount of data to be sorted which can't be placed in main memory at a time, then the data that is currently being sorted is brought into main memory and rest is on secondary memory i.e. on external files like disks and tapes. This type of sorting is called external sorting.

In internal sort all the data is in main memory so it is easy to access any element while it is not so in external sort. In this chapter we will discuss internal sorting only.

8.3 Sort Stability

Sort stability comes into picture if the key on which data is being sorted is not unique for each record, i.e. two or more records have identical keys. For example consider a list of records where each record contains name and age of a person. We will take name as the sort key and sort all the records according to the names. The unsorted list of these records is given in the first table while the next three tables have sorted list.

Name	Age
Vineet	25
Amit	37
Deepa	67
Shriya	45
Deepa	20
Kiran	18
Deepa	56

Unsorted list

Name	Age
Amit	37
Deepa	56
Deepa	67
Deepa	20
Kiran	18
Shriya	45
Vineet	25

Sorted list (Unstable Sort)

Name	Age
Amit	37
Deepa	20
Deepa	56
Deepa	67
Kiran	18
Shriya	45
Vineet	25

Sorted list (Unstable Sort)

Name	Age
Amit	37
Deepa	67
Deepa	20
Deepa	56
Kiran	18
Shriya	45
Vineet	25

Sorted list (Stable Sort)

Figure 8.2 Stable and Unstable Sort

Any sorting algorithm would place (Amit,37) in first position, (Kiran,18) in fifth position, (Shriya,45) in sixth position and (Vineet,25) in seventh position. There are three records with identical keys(names), which are (Deepa,67), (Deepa,20) and (Deepa,56) and any sorting algorithm would place them in adjacent locations i.e. second, third and fourth locations but not necessarily in the same relative order.

A sorting algorithm is said to be stable if it maintains the relative order of the duplicate keys in the sorted output i.e. if keys are equal then their relative order in the sorted output is same. For example if records R_i and R_j have equal keys and if record R_i precedes record R_j in the input data then R_i should precede R_j in the sorted output data also if the sort is stable. If the sort is not stable then R_i and R_j may be in any order in the sorted output. So in an unstable sort the duplicate keys may occur in any order in the sorted output.

In our example the first two sorted lists did not maintain the relative order of the duplicate keys while the ird one did. So we can say that the first two sorted lists were obtained by unstable sorting algorithms while the st one was obtained by a stable sorting algorithm.

Sometimes we need to sort the records according to different keys at different times i.e. records which are rted on one key are again sorted on another key. In these types of situations an unstable sort is not desirable. et us take an example and see why it is so.

Suppose we have a list of all students of a school consisting of their names and classes, and the list is phabetically sorted on name, i.e. all the names are in alphabetical order. Now suppose we want to sort this list ith respect to class. Any sorting algorithm will place the names of all classmates in adjacent locations, but ly a stable sort will place the names of students in a particular class alphabetically.

Name	Class
Amit	12
Anuj	11
Chetan	10
Deepak	12
Karan	12
Manav	11
Neelam	10
Raj	11

List L with names in sorted order

Name	Class
Neelam	10
Chetan	10
Manav	11
Raj	11
Anuj	11
Deepak	12
Amit	12
Karan	12

List L sorted on class (Unstable Sort)

Name	Class
Chetan	10
Neelam	10
Anuj	11
Manav	11
Raj	11
Amit	12
Deepak	12
Karan	12

List L sorted on class (Stable Sort)

Figure 8.3

e can see that in stable sort we got the names of students of each class in alphabetical order while unstable rt disturbed the initial order of students who were in same class.

.4 Sort by Address(Indirect Sort)

rting can be done in two ways, by actually moving the records or by maintaining an auxiliary array of inters and rearranging the pointers in that array. Let us first see how the sorting is done by moving the cords.

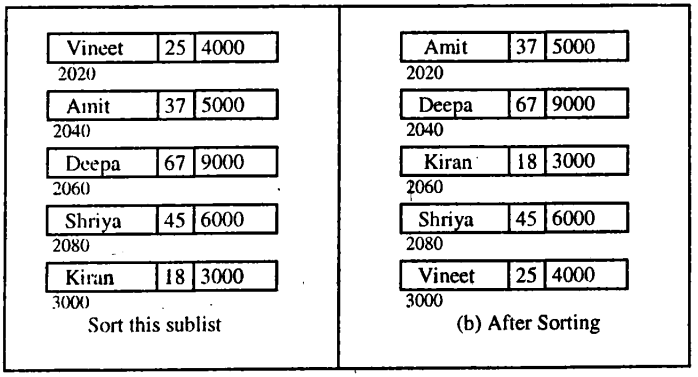


Figure 8.4 Sorting by moving Records

Here we can see that the records are moved from one place to another in the memory, for example the record incet, 25, 4000) was initially stored at address 2020 but after sorting it is at address 3000.

If records to be sorted are very large then this process of moving records can be an expensive task. In this se we can take an array of pointers, which contains addresses of the records in memory. Now instead of rranging the records, we rearrange the addresses inside the pointer array. In figure 8.5, we have performed ting on the same records but this time by adjusting pointers in the pointer array.

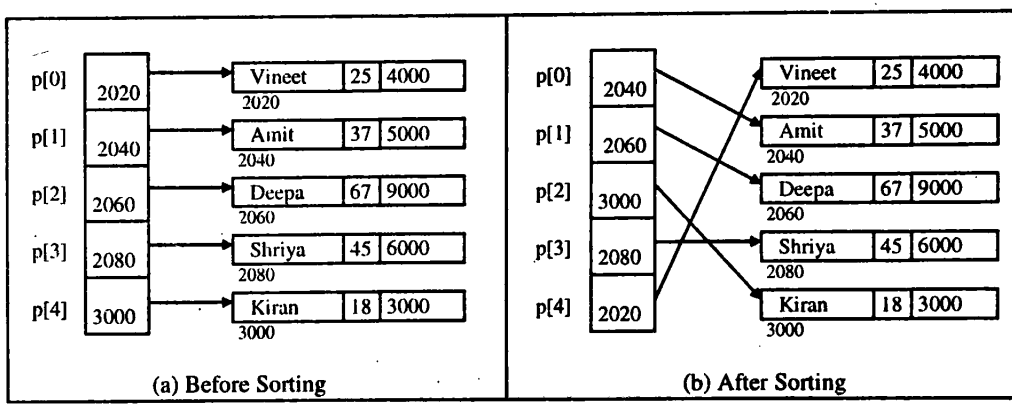


Figure 8.5 Sorting by Rearranging Pointers

Here we can see that records are at the same position but the pointers, which are pointing to records, are in different sequence but still pointing to the same record. For example, the record (Vineet, 25, 4000) was stored at address 2020 before and after the sorting. Before sorting, the first pointer of the array was pointing to first record, second pointer to the second record and so on. After sorting, first pointer will point to the first record of sorted list, second pointer will point to the second record of sorted list and so on.

This process of sorting by adjusting pointers is called sorting by address or indirect sort because we can indirectly refer to the records in sorted order. If the records are elements of an array then we can store the indices of these elements in an auxiliary array instead of storing the addresses and sorting can be done indirectly by rearranging the values of indices contained inside the auxiliary array.

In the programs that we take in this chapter, we will perform the sorting by moving the data and the extension of these programs to perform sorting by address is simple. After the discussion of all sort methods, we have given a program that illustrates sorting by address(P8.16).

Another way to avoid movement of records while sorting is to store the records in a linked list. In that case we will have to only change the pointers instead of moving the records. Here we have overhead of an extra pointer field but if the records are very large then this overhead is negligible.

8.5 In place Sort

In place sorting methods generally use the same storage that is occupied by input data to store the output data while sorting. These methods do not need any extra storage except possibly a very little amount of working storage that can be neglected. So the additional space requirements of these types of sorts are O(1).

Other sorting methods may need extra storage to store intermediate results of the sorting, and at last the sorted data is copied back to original storage. In these methods the amount of extra storage needed is proportional to the size of data.

For example suppose we have to sort an array of n elements and we need another array of size n to perform this sort then we are not doing an in place sort. If we need only constant number of extra variables, i.e. number of variables required is independent of the size of array then the sort is in place sort. The in place sorts are also known as minimal storage sorting methods. If the size of array to be sorted is very large then it is beneficial to use an in place sort.

Merge sort is not an in place sort because it requires an extra array of size n to sort an array of size n. Selection sort, bubble sort, insertion sort methods are in place sort methods.

8.6 Sort pass

›rocess of sorting requires traversing the given list many times. These traversals may be on the whole list ›art of it depending on the algorithm. This procedure of sequentially traversing the list or a part of it is l a pass. Each pass can be considered as a step in sorting and after the last pass we get the sorted list.

Sort Efficiency

ıg is an important and frequent operation in many applications and so the aim is not only to get the sorted ›ut to get it in most efficient manner. Therefore many algorithms have been developed for sorting and to e which one to use we need to compare them using some parameters. The choice is made using these three ıeters -

ding time
ace requirement
n time or execution time

data is in small quantity and sorting is needed only at few occasions then any simple sorting technique l be adequate. This is because in these cases a simple or less efficient technique would behave at par with omplex techniques developed to minimize run time and space requirements. So there is no point in ing lot of time in searching for best sorting algorithm or implementing a complicated technique.

e have already discussed about the space requirement of a sort, and we've seen that if data to be sorted is ge quantity then it is better to use an in place sort.

ıe most important parameter is the running time of the algorithm. If the amount of data to be sorted is in quantity, then it is crucial to minimize run time by choosing an efficient sorting technique.

ıe two basic operations in any sorting algorithm are comparisons and record movements. The record s or any other operations are generally a constant factor of the number of comparisons and moreover the l moves can be considerably reduced, so the run time is calculated by measuring the number of arisons. Calculating the exact number of comparisons may not be always possible so an approximation is by big-O notation. Thus the run time efficiency of different algorithms is expressed in terms of O on. The efficiency of most of the sorting algorithms is in between O(n log n) and $O(n^2)$.

some sorting algorithms, the time taken to sort depends on the order in which elements appear in the al data i.e. these algorithms behave differently when the data is already sorted or when it is in reverse For example if the data to be sorted is {4, 6, 8, 9, 10}, then an intelligent algorithm will immediately find at the data is already sorted and it will not waste time in doing anything. Some sorting algorithms always ame time to sort, irrespective of the order of data.

e run time of a data sensitive algorithm may be different for different orders of data; hence we need to ze the sorting algorithms in three different cases which are -

›ut data is in sorted order(ascending), e.g. { 1, 2, 3, 4, 5, 6, 7, 8 }

›ut data is in random order, i.e. all the elements are dispersed in the data and there is no specific order g these elements e.g. { 4, 8, 1, 6, 5, 2, 3, 7 }. In this case it is assumed that all n! permutations of data are y likely where n is size of data.

›ut data is in reverse sorted order(descending) e.g. { 8, 7, 6, 5, 4, 3, 2, 1}.

ere are numerous sorting algorithms but none of them can be termed best or most efficient, each algorithm : advantages and disadvantages. The choice of a sorting algorithm depends on the specific situation. For ›le if we know in advance that our data is almost sorted then it would be useful to use an algorithm which entify this order. The size of data is also considered while deciding which algorithm to choose. The ıt of space available also determines our choice of algorithm. If we have no extra space then we have to in place algorithms. So we can see that the implementation of particular sorting technique depends not n the order of that technique but also on the situation and type of data. Now after this introduction of g we are ready to study various sorting algorithms and their analysis.

Selection Sort

Suppose that you are given some numbers and asked to arrange them in ascending order. The most intuiti way to do this would be to find the smallest number and put in the first place and then find the second smalle number and put it in the second place and so on. This is the simple technique on which selection sort is based. is named so because in each pass it selects the smallest element and keeps it in its exact place.

Suppose we have n elements stored in an array `arr`. First we will search the smallest element fro `arr[0].....arr[n-1]` and exchange it with `arr[0]`. This will place the smallest element of list at position of array. Now we will search smallest element from remaining elements `arr[1].....arr[n-1]` a exchange it with `arr[1]`. This will place the second smallest element of the list at 1st position of array. Th process continues till the whole array is sorted. The whole process is as-

Pass 1 :
1. Search the smallest element from arr[0].....arr[n-1].
2. Exchange this element with arr[0].

Result : arr[0] is sorted.

Pass 2 :
1. Search the smallest element from arr[1].....arr[n-1].
2. Exchange this element with arr[1].

Result : arr[0], arr[1] are sorted.

.....

.....

Pass n-1 :
1. Search the smallest element from arr[n-2] and arr[n-1].
2. Exchange this element with arr[n-2].

Result : arr[0], arr[1],.....arr[n-2] are sorted.

Now all the elements except the last one have been put in their proper place. The remaining last eleme `arr[n-1]` will definitely be the largest of all and so it is automatically at its proper place. So we need only n- passes to sort the array. Let us take a list of elements in unsorted order and sort it by applying selection sort.

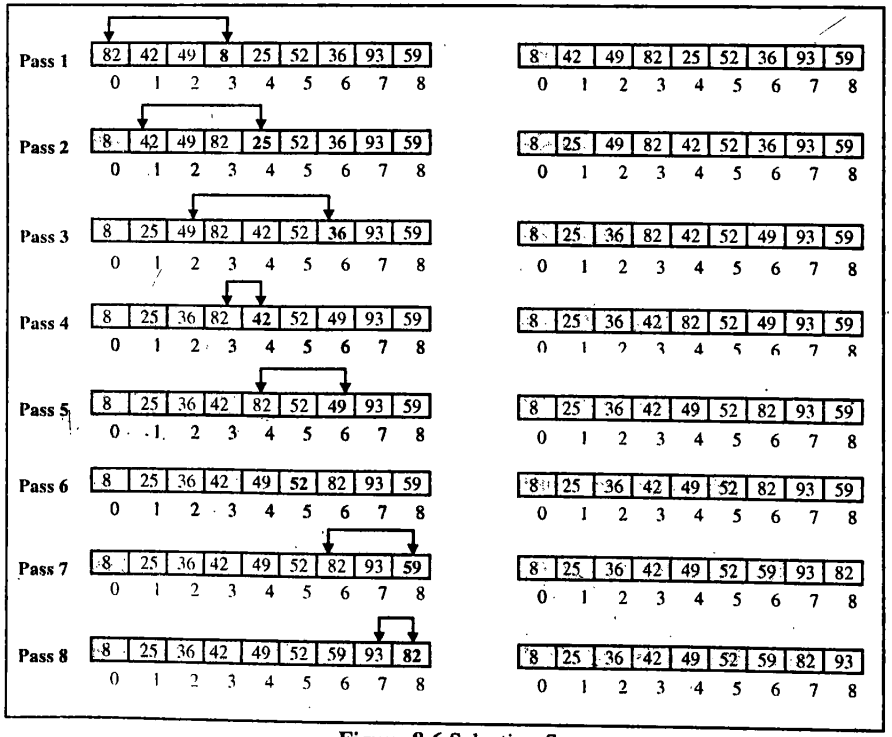


Figure 8.6 Selection Sort

In the first pass, 8 is the smallest element among arr[0].....arr[8], so it is exchanged with arr[0] 82. In the second pass, 25 is the smallest among arr[1].....arr[8] so it is exchanged with arr[1] i.e. Similarly other passes also proceed. The shaded portion shows the elements that have been put in their final ace.

```
/P8.1. Program of sorting using selection sort*/
nclude <stdio.h>
efine MAX 100
in()

    int arr[MAX],i,j,n,temp,min;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d", &arr[i]);
    }
    /*Selection sort*/
    for(i=0; i<n-1; i++)
    {
        /*Find the index of smallest element*/
        min = i;
        for(j=i+1; j<n; j++)
        {
            if(arr[min] > arr[j])
                min = j ;
        }
        if(i!=min)
        {
            temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp ;
        }
    }
    printf("Sorted list is : \n");
    for(i=0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
*End of main()*/
```

Each iteration of outer for loop corresponds to a single pass. In each iteration of outer for loop we have to change arr[i] with the smallest element among arr[i]...arr[n-1]. The inner for loop is used to find the dex of the smallest element and it is stored in min. Initially variable min is initialized with i. After this, r[min] is compared with each of the elements arr[i+1], arr[i+2].....arr[n-1] and whenever we get maller element, its index is assigned to min.

After finding the smallest element, it is exchanged with arr[i]. We have preceded this swap operation with condition to avoid swapping of an element with itself. This situation arises when an element is already in its oper place. In the pass 6 of figure 8.6, arr[5] has to be swapped with arr[5] which is obviously redundant.

8.1 Analysis of Selection Sort

selection sort, the number of comparisons does not depend on the order of data i.e. selection sort is not data nsitive. The number of comparisons performed is same whether input data is sorted, reverse sorted or in ndom order. In first pass, arr[0] is compared with arr[1].....arr[n-1] so there will be n-1 mparisons, in second pass arr[1] is compared with arr[2].....arr[n-1] so there will be n-2 mparisons. In the last pass, arr[n-2] is compared with arr[n-1] so there will be only one comparison. e total number of comparisons will be-

$(n-1) + (n-2) + (n-3) + \ldots\ldots\ldots\ldots + 3 + 2 + 1 = n(n-1)/2 = O(n^2)$

So the efficiency of selection sort is $O(n^2)$ in all the three cases.

Selection sort is simple to implement and requires only one temporary variable for swapping elements. Th main advantage of selection sort is that data movement is very less. If an element is at its proper place then will not be moved at all, so if many elements are at proper place i.e. the list is almost sorted then there will b very little data movement. Thus we see that the number of swaps depends on the order of data, and it can nev be more than `n-1`. The swaps are very less as compared to insertion and bubble sorts. If the records are larg then cost of moving data is more than the cost of comparisons, so if the records are large it is better to us selection sort.

Selection sort is not a stable sort. It requires only one temporary variable so it is an in-place sort, spac complexity is O(1).

8.9 Bubble Sort

Bubble sort proceeds by scanning the list and exchanging the adjacent elements if they are out of order wi respect to each other. It compares each element with its adjacent element and swaps them if they are not i order i.e. `arr[i]` will be compared with `arr[i+1]` and if `arr[i]>arr[i+1]` then they will be swapped.

In selection sort we searched for the smallest element and then performed the swap, while here swap will b performed as soon as we find two adjacent elements out of order. So in selection sort there was only one swa in a pass while in bubble sort there may be many swaps in a single pass. Hence bubble sort is not an efficie sorting technique but it is simple and easy to implement.

After first pass the largest element will be at its proper position in the array i.e. $(n-1)^{th}$ position, aft second pass the second largest element will be placed at its proper position i.e. $(n-2)^{th}$ position. Similar after each pass the next larger elements will be moved to the end of the list and placed at their proper position If there are `n` elements, then only `n-1` passes are required to sort the array and the procedure is as-

Pass 1 :
Compare arr[0] and arr[1], If arr[0] > arr[1] then exchange them,
Compare arr[1] and arr[2], If arr[1] > arr[2] then exchange them,
Compare arr[2] and arr[3], If arr[2] > arr[3] then exchange them,
.....
.....
Compare arr[n-2] and arr[n-1], If arr[n-2] > arr[n-1] then exchange them.
Result : Largest element is placed at $(n-1)^{th}$ position
arr[n-1] is sorted.

Pass 2 :
Compare arr[0] and arr[1], If arr[0] > arr[1] then exchange them,
Compare arr[1] and arr[2], If arr[1] > arr[2] then exchange them,
Compare arr[2] and arr[3], If arr[2] > arr[3] then exchange them,
.....
Compare arr[n-3] and arr[n-2], If arr[n-3] > arr[n-2] then exchange them.
Result : Second largest element is placed at $(n-2)^{th}$ position
arr[n-2], arr[n-1] are sorted.

.....
.....

Pass n-2 :
Compare arr[0] and arr[1], If arr[0] >arr[1] then exchange them,
Compare arr[1] and arr[2], If arr[1] >arr[2] then exchange them.
Result : arr[2],.....arr[n-2], arr[n-1] are sorted.

Pass n-1 :
Compare arr[0] and arr[1], If arr[0] >arr[1] then exchange them.

Result : arr[1], arr[2],.....arr[n-2], arr[n-1] are sorted.

In the second pass, comparisons will be done only up to $(n-2)^{th}$ position i.e. the last comparison is done between `arr[n-3]` and `arr[n-2]`, because the largest element has already been placed at its proper position i.e. position `(n-1)`. In the third pass, the last comparison is done between `arr[n-4]` and `arr[n-3]`, because the second largest element has already been placed at its proper position i.e. position `(n-2)`. In the last pass there is only one comparison which is in between `arr[0]` and `arr[1]`. The remaining element `arr[0]` will definitely be the smallest element, so the whole list is sorted after `n-1` passes.

Let us take an array in unsorted order and sort it by applying bubble sort.

40 20 50 60 30 10

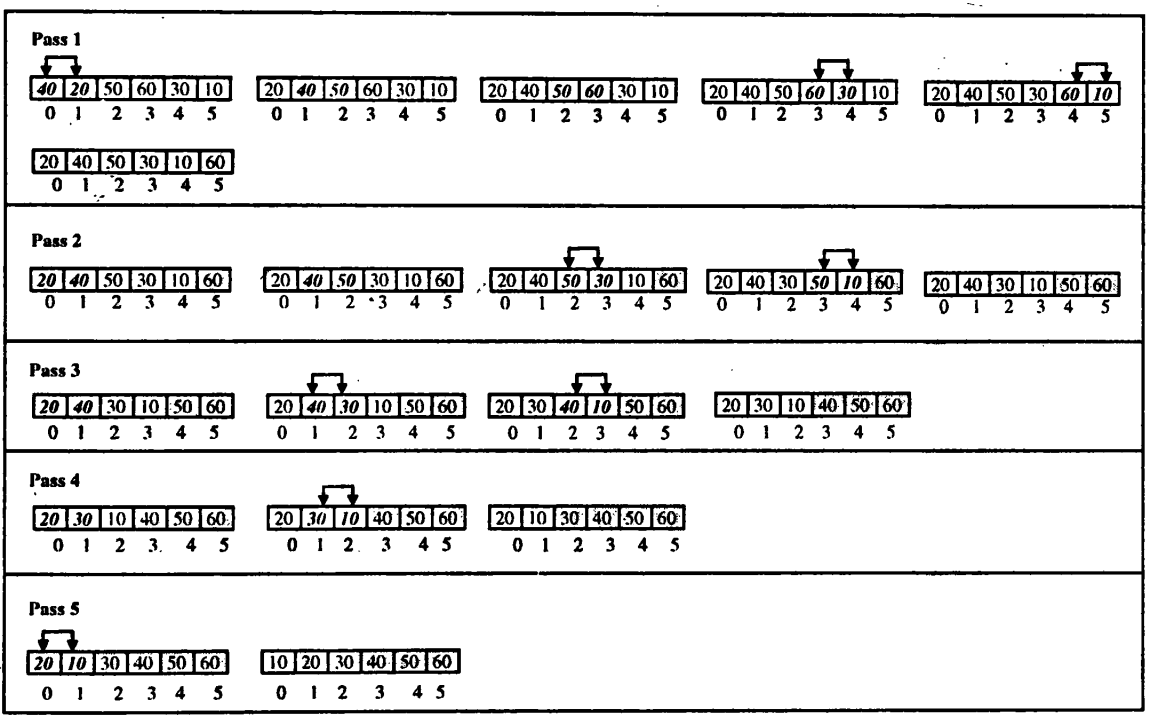


Figure 8.7 Bubble sort

Let us see what happens in the first pass. First `arr[0]` is compared with `arr[1]`, since 40>20 they are swapped. Now `arr[1]` is compared with `arr[2]`, since 40<50 they are not swapped. Now `arr[2]` is compared with `arr[3]`, since 50<60 they are not swapped. Now `arr[3]` is compared with `arr[4]`, since 60>30 they are swapped. Now `arr[4]` is compared with `arr[5]`, since 60>10 they are swapped. At the end of this pass the largest element 60 is placed at the last position. The elements which are being compared are shown in italics and the elements which have been placed in proper place are shaded.

Sometimes it is possible that a list of n elements becomes sorted in less than `n-1` passes. For example consider this list - 40 20 10 30 60 50

After first pass the list is - 20 10 30 40 50 60

After second pass the list is - 10 20 30 40 50 60

The list of 6 elements becomes sorted in only 2 passes. Hence other passes are unnecessary and there is no need to proceed further. Now the question is how we will be able to know that the list has become sorted. If no swaps occur in a pass it means that the list is sorted. For example in the above case there will be no swaps in the

third pass. We can take a variable that keeps record of the number of swaps in a pass and if no swaps occur then we can terminate our procedure.

```
/*P8.2 Program of sorting using bubble sort*/
#include <stdio.h>
#define MAX 100
main()
{
        int arr[MAX],i,j,temp,n,xchanges;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        /*Bubble sort*/
        for(i=0; i<n-1 ;i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(arr[j] > arr[j+1])
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges==0) /*If list is sorted*/
                        break;
        }
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
        printf("\n");
}/*End of main()*/
```

The figure 8.8 shows the procedure of bubble sort along with the values of `i` and `j`. Each iteration of outer for loop corresponds to a single pass.

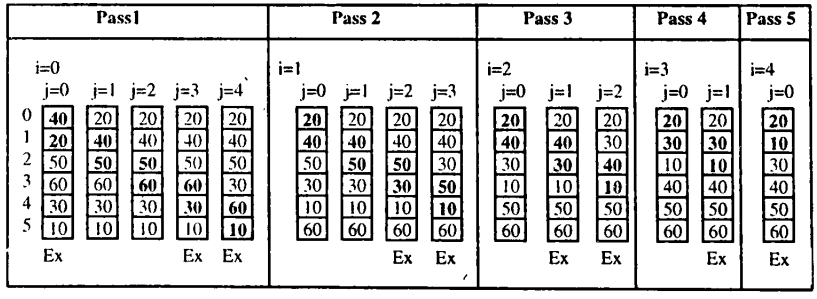


Figure 8.8 Bubble Sort

8.9.1 Analysis of Bubble Sort

The number of passes will depend on the order of data. There can be minimum 1 pass and maximum `(n-1)` passes.

8.9.1.1 Data in sorted order

If all the elements are sorted then only one pass is required and so there will be only one iteration of outer for loop. The number of comparisons will be (n-1) and all elements are in their proper place so there will be no swaps. Hence the time complexity in this case is O(n).

8.9.1.2 Data in reverse sorted order

If the array elements are in reverse order, (n-1) passes are required and so there will be (n-1) iterations of the outer for loop. In first iteration there will be (n-1) comparisons, in second iteration there will be (n-2) comparisons, in third iteration there will be (n-3) comparisons and so on. So the total number of comparisons is

$(n-1) + (n-2) + (n-3) + \ldots\ldots\ldots + 3 + 2 + 1$
$= n(n-1)/2 = O(n^2)$
The total number of swaps will be equal to the number of comparisons.

8.9.1.3 Data in random order

In the i^{th} iteration the number of comparisons is n-i. So if p passes are required to sort the data, then the total number of comparisons would be -

$(n-1) + (n-2) + (n-3) + \ldots\ldots\ldots (n-p)$
$= p/2[2(n-1) + (p-1)(-1)]$
$= (2pn - p^2 - p])/ 2$

It can be shown that the number of passes p in average case is O(n) so the comparisons will be $O(n^2)$. The main advantage of this algorithm is that it is simple and easy to implement, additional space requirement is only one temporary variable and it behaves O(n) for sorted array of elements. Bubble sort should not be used for large lists; it should generally be used for smaller lists.

Bubble sort is a stable sort. It requires only one temporary variable so it is an in place sort, space complexity is O(1).

8.10 Insertion Sort

The insertion sort proceeds by inserting each element at the proper place in a sorted list. This is the same technique used by card players for arranging cards. When they receive a card, they place it in the appropriate place among the cards that they have already arranged.

We will consider our list to be divided into two parts - sorted and unsorted. Initially the sorted part contains only the first element of the list and unsorted part contains the rest of the elements. In each pass, the first element from the unsorted part is taken and inserted into the sorted part at appropriate place. If there are n elements in the list, then after n-1 passes the unsorted part disappears and our whole list becomes sorted. The process of inserting each element in proper place is as-

Pass 1 :
Sorted part : arr[0]
Unsorted part : arr[1], arr[2],arr[3], ... ,arr[n-1]
arr[1] is inserted before or after arr[0].
Result : arr[0] and arr[1] are sorted.
Pass 2 :
Sorted part : arr[0], arr[1]
Unsorted part : arr[2],arr[3], ... ,arr[n-1]
arr[2] is inserted before arr[0], or in between arr[0] and arr[1] or after arr[1].
Result : arr[0], arr[1] and arr[2] are sorted.
Pass 3 :
Sorted part : arr[0], arr[1],arr[2]

Unsorted part : arr[3], arr[4],..... ,arr[n-1]
arr[3] is inserted at its proper place among arr[0], arr[1], arr[2]
Result : arr[0], arr[1], arr[2], arr[3] are sorted.
.....
.....

Pass n-1 :
Sorted part : arr[0], arr[1], ...arr[n-2]
Unsorted part : arr[n-1]
arr[n-1] is inserted at its proper place among arr[0], arr[1], ...arr[n-2]
Result : arr[0], arr[1],arr[3],.....arr[n-1] are sorted.

To insert an element in the sorted part we need a vacant position, and this space is created by moving all the larger elements one position to the right. Now let us take a list of elements in unsorted order and sort them by applying insertion sort.

82 42 49 8 25 52 36 93 59

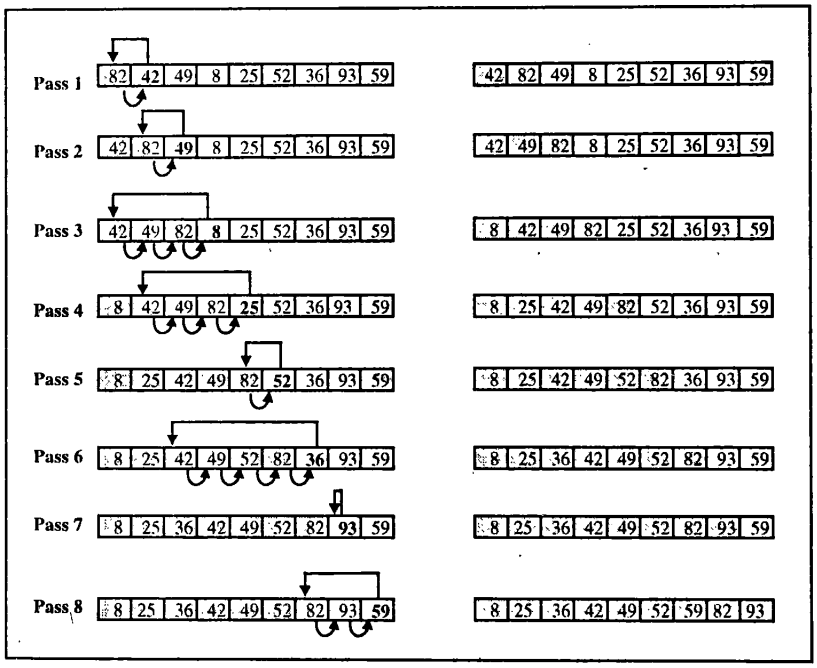


Figure 8.9 Insertion Sort

The shaded portion shows the sorted part of the list and the arrows show the movement of elements. To insert an element we have to scan the sorted part and search for the appropriate place where this element can be inserted. We will use sequential search for this purpose. We also need a vacant position for inserting this element, and for this all the larger elements are moved one position right. For example in pass 6 the elements 42, 49, 52, 82 are moved one position right to make space for insertion of 36.

If we start our searching from the end of the sorted part instead of its beginning then we can simultaneously shift the elements also.

```
/*P8.3 Program of sorting using insertion sort*/
#include<stdio.h>
#define MAX 100
main()
{
        int arr[MAX],i,j,k,n;
```

```
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d", &arr[i]);
        }
        /*Insertion sort*/
        for(i=1; i<n; i++)
        {
                k=arr[i]; /*k is to be inserted at proper place*/
                for(j=i-1; j>=0 && k<arr[j]; j--)
                        arr[j+1]=arr[j];
                arr[j+1]=k;
        }
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
        printf("\n");
}/*End of main()*/
```

In each iteration of for loop, the first element of the unsorted part is inserted into the sorted part. The element to be inserted (`arr[i]`) is stored in the variable `k`. In the inner for loop, the sorted part is scanned to find the exact location for the insertion of the element `arr[i]`. The search starts from the end of the sorted part so variable `j` is initialized to `i-1`. The search stops when we either reach the beginning of the sorted part or we get an element less than `k`. Inside the inner for loop, the elements are moved right one position, and obviously these are elements which are greater than `k`. At the end `k` is inserted at its proper place.

8.10.1 Analysis of Insertion sort

The outer for loop will always have `n-1` iterations. The iterations of the inner for loop will vary according to the data. For each iteration of outer for loop, the inner for loop executes 0 to `i` times, i.e. the inner for loop can be executed minimum 0 times and maximum `i` times.

arr[i] has to be put in any of these i+1 positions 0,1,2,3....., i-1, i

* If arr[i] is already at proper place i.e. position i is the proper place for arr[i]
  1 comparison, arr[i] compared with arr[i-1]
  No move inside inner for loop and 2 moves for loading and unloading k
* If arr[i] has to be placed at position i-1
  2 comparisons, arr[i] compared with arr[i-1], arr[i-2]
  1 move inside inner for loop and 2 moves for loading and unloading k
* If arr[i] has to be placed at position i-2 ,
  3 comparisons, arr[i] compared with arr[i-1], arr[i-2],arr[i-3]
  2 moves inside inner for loop and 2 moves for loading and unloading k

  .....
  .....

* If arr[i] has to be placed at position 2
  i-1 comparisons, arr[i] compared with arr[i-1], arr[i-2],arr[i-3],.....,arr[1]
  i-2 moves inside inner for loop and 2 moves for loading and unloading k
* If arr[i] has to be placed at position 1,
  i comparisons, arr[i] compared with arr[i-1], arr[i-2],arr[i-3],.....,arr[1],arr[0]
  i-1 moves inside inner for loop and 2 moves for loading and unloading k
* If arr[i] has to be placed at position 0
  i comparisons, arr[i] compared with arr[i-1], arr[i-2],arr[i-3],.....,arr[1],arr[0]
  i moves inside inner for loop and 2 moves for loading and unloading k

8.10.1.1 Data in sorted order

The best case occurs if the list is in sorted order. In each iteration of outer for loop, `arr[i]` is always in proper place. We have seen that if `arr[i]` is at the proper place then there is only one comparison. So there will be only one comparison in each iteration of the outer for loop, and the outer for loop always iterates `n-1` times so there will be total `n-1` comparisons in all, which is O(n).

The body of inner for loop will never be entered, so in each iteration of outer for loop there will be only 2 moves and hence the total number of moves will be `2(n-1)`. All these moves are unnecessary.

8.10.1.2 Data in reverse sorted order

The worst case occurs when the list is sorted in reverse order. In any i^{th} iteration of outer for loop, the element `arr[i]` will be less than each of these elements `arr[0]`, `arr[1]`, ...`arr[i-1]`, so `arr[i]` will always be inserted at 0^{th} position. We have seen above that if the element `arr[i]` has to be placed at 0^{th} position then there will be `i` comparisons. So in first iteration of outer for loop, there will be 1 comparison, in second iteration of outer for loop there will be 2 comparisons and in `(n-1)` th iteration of outer for loop there will be `n-1` comparisons. So the total number of comparisons will be-

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + ... + (n-1) = n(n-1)/2 = O(n^2)$$

If `arr[i]` is to be placed at 0^{th} position then the number of moves in i^{th} iteration of for loop is `i+2`, the total number of moves will be-

$$\sum_{i=1}^{n-1} i+2 = n(n-1)/2 + 2(n-1) = O(n^2)$$

There is a variation in the above two cases so now let us analyze the efficiency of insertion sort when data is in random order.

8.10.1.3 Data in random order

Here we assume equal probability of occupying all array locations.

Average number of comparisons in i^{th} iteration of outer for loop

$$\frac{1+2+3+....+(i-1)+i+i}{i+1}$$

$$= [\, i(i+1)] \,/\, 2(i+1) + i/(i+1)$$

$$= i/2 + 1 - 1/(i+1)$$

Total number of comparisons

$$= \sum_{i=1}^{n-1} i/2 + \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-1} 1/(i+1)$$

$$= n(n-1)/4 + (n-1) - \sum_{i=1}^{n-1} 1/(i+1)$$

$$= n(n-1)/4 + n - \sum_{j=1}^{n} 1/j$$

$$\approx n(n-1)/4 + n - \log_e n$$

$$= O(n^2)$$

Average number of moves in i^{th} iteration of outer for loop

$$\frac{0+1+2+\ldots..+(i-1)+i}{(i+1)}$$
$= [i(i+1)] / [2(i+1)]$
$= i/2$
Total number of moves
$= \sum_{i=1}^{n-1} i/2 = n(n-1)/4$
$= O(n^2)$

The advantage of insertion sort is its simplicity, and it is very efficient when number of elements to be sorted are very less. because for smaller file size n the difference between $O(n^2)$ and
$O(n \log n)$ is very less and $O(n \log n)$ generally requires complex sorting technique. Insertion sort is also efficient for lists that are almost sorted.

We can simplify the inner for loop, or reduce a condition in inner for loop, by taking a sentinel value in `arr[0]`. This value is taken to be smaller than the smallest value possible. So we can avoid the test `j>=0`. This condition is true only when the element to be inserted is the smallest and has to be inserted in the beginning of the array. The elements to be sorted can be stored in `arr[1]` to `arr[n]` and a sentinel value in `arr[0]`. So now only single test is sufficient and this will simplify the inner loop.

We can reduce the number of comparisons by using binary search for finding the proper place of insertion. But still the elements have to be shifted right one position which is $O(n^2)$. So the use of binary search would not improve the efficiency of the insertion sort.

A disadvantage of this sorting is the number of movements. The elements of the sorted part also have to move to make place for another element. When the data items to be sorted are large then these movements can prove costly.

Sometimes elements are at their proper place in the list but still they are moved. For example consider this data - 75 19 34 55 12
Here 19, 34 and 55 are at their proper places so their movement is redundant.

Insertion sort is a stable sort. It requires only one temporary variable so it is an in-place sort, space complexity is O(1).

8.11 Shell Sort (Diminishing Increment Sort)

The insertion sort is not efficient because many moves are performed, and the reason for so many moves is that elements are moved only one position at a time. The elements are moved only one position because only adjacent elements are compared. If we can compare elements that are far apart then we can considerably reduce the number of moves thereby making insertion sort more efficient. This is what happens in Shell sort which is an improved version of insertion sort and was given by Donald. L. Shell in 1959. It works by first comparing elements that are far apart and then it compares closer elements, the distance between the elements to be compared reduces with every pass until the last pass where adjacent elements are compared.

Now let us see the procedure of this sorting technique. In each pass we take a number 'h' called the increment. This number decreases in subsequent passes and finally in the last pass it is always 1. In each pass we divide the list into h sublists. Each sublist is created by taking elements that are at a distance of h from each other. For example if we have an array of 17 elements and we take the increments as 5, 3, 1 then the sublists will be created as-

Pass 1: (increment = 5)
5 sublists are created
Sublist 1 : a[0], a[5], a[10], a[15]
Sublist 2 : a[1], a[6], a[11], a[16]
Sublist 3 : a[2], a[7], a[12]
Sublist 4 : a[3], a[8], a[13]

Sublist 5 : a[4], a[9], a[14]
Pass 2 : (increment = 3)
3 sublists are created
Sublist 1 : a[0], a[3], a[6], a[9], a[12], a[15]
Sublist 2 : a[1], a[4], a[7], a[10], a[13], a[16]
Sublist 3 : a[2], a[5], a[8], a[11], a[14]
Pass 3 : (increment =1)
1 sublist is created
Sublist 1 : a[0], a[1], a[2], a[3], a[4], a[5], a[6],a[7], a[8], a[9], a[10], a[11], a[12], a[13], a[14], a[15], a[16]

These sublists are separately sorted among themselves by insertion sort and at the end of the pass these sorted sublists are combined. The list that we get after each pass is partially sorted. In the last pass, increment is 1 so only one sublist is created which consists of all the elements, so insertion sort is performed on the whole list and we get our sorted list.

Since the value of increment continually decreases, this sort is also known as diminishing increment sort.Now we will take an array of 17 elements and see how they can be sorted by shell sort if the increments are taken to be 5, 3, 1. The sorting procedure is shown in figure 8.10.

In the first pass, 5 sublists are created and these sublists are sorted among themselves using insertion sort. For example the first and second sublists are { 19, 10 , 45 ,12 } and { 63, 1, 3, 56 } and after sorting they become { 10, 12, 19 , 45 } and { 1, 3, 56, 63}. Similarly other 3 sublists are also sorted and then all the five sorted sublists are combined. The list that we get after first pass is partially sorted i.e. all the elements that are at a distance of 5 from each other are sorted. This list is also called 5-sorted list and the procedure is called 5-sort. Similarly in the second pass 3-sort is performed and we get 3-sorted list. In the last pass, 1-sort is done and we get our sorted list.

Original list	19	63	2	6	7	10	1	18	9	4	45	3	5	17	16	12	56
Pass 1 : Partiton into 5 sublists	19 – – – –	– 63 – – –	– – 2 – –	– – – 6 –	– – – – 7	10 – – – –	– 1 – – –	– – 18 – –	– – – 9 –	– – – – 4	45 – – – –	– 3 – – –	– – 5 – –	– – – 17 –	– – – – 16	12 – – – –	– 56 – – –
Sort above 5 sublists	10 – – – –	– 1 – – –	– – 2 – –	– – – 6 –	– – – – 4	12 – – – –	– 3 – – –	– – 5 – –	– – – 9 –	– – – – 7	19 – – – –	– 56 – – –	– – 18 – –	– – – 17 –	– – – – 16	45 – – – –	– 63 – – –
Combine these 5 sorted sublists	10	1	2	6	4	12	3	5	9	7	19	56	18	17	16	45	63
Pass 2 : Partition into 3 sublists	10 – –	– 1 –	– – 2	6 – –	– 4 –	– – 12	3 – –	– 5 –	– – 9	7 – –	– 19 –	– – 56	18 – –	– 17 –	– – 16	45 – –	– 63 –
Sort above 3 sublists	3 – –	– 1 –	– – 2	6 – –	– 4 –	– – 9	7 – –	– 5 –	– – 12	10 – –	– 17 –	– – 16	18 – –	– 19 –	– – 56	45 – –	– 63 –
Combine these 3 sorted sublists	3	1	2	6	4	9	7	5	12	10	17	16	18	19	56	45	63
Pass 3 : Only one sublist	3	1	2	6	4	9	7	5	12	10	17	16	18	19	56	45	63
	1	2	3	4	5	6	7	9	10	12	16	17	18	19	45	56	63

Figure 8.10 Shell Sort

Initially the elements which are far apart are compared and then the elements which are closer and so on. ıe distance between elements being compared decreases with each pass and finally in the last pass adjacent ements are compared. This way we have improved over insertion sort because here elements can be moved ng distances instead of only one place at a time.

We know that if the list is in almost sorted order then insertion sort proves to be very efficient. Another ature of insertion sort is that it is very efficient on small lists. These two aspects of insertion sort are the basis Shell sort.

Initially when the value of increment is large, the size of sublists is small so insertion sort is fast. After each ıss the elements move closer to their final positions i.e. the list becomes more nearly sorted after each pass. ıe value of increment decreases with each pass hence leading to larger sublists, and insertion sort is not ıitable for large lists. But insertion sort works fast on these larger sublists also because of the fact that they are :arly sorted. When we reach the last pass the list becomes almost sorted i.e. the elements are very much close their final positions, so the insertion sort on the whole list goes very rapidly. So we can see that in Shell sort e previous passes help in making the list almost sorted so that the insertion sort in the last pass is very fast.

Now let us talk about the choice of increments. There is no restriction on sequence of increments except that should be 1 in the last pass. The increment sequence suggested by Shell originally was to take first increment ıual to half the size of list and divide the increment value by 2 in each pass.

If we have a list of 17 elements then the increment sequence will be 8, 4, 2, 1. The sublists formed in this ıse are given below (only subscripts are shown).

crement = 8 { 0, 8, 16 }, { 1, 9 }, { 2, 10 }, { 3, 11 }, { 4, 12 }, { 5, 13 }, {6, 14}, { 7, 15 }
crement = 4 { 0, 4, 8, 12, 16 }, { 1, 5, 9, 13 }, { 2, 6, 10, 14 }, { 3, 7, 11, 15 }
crement = 2 { 0, 2, 4, 6, 8, 10, 12, 14, 16 }, { 1, 3, 5, 7, 9, 11, 13,15 }
crement = 1 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 }

We can see that elements in the even and odd positions will not be compared till the last pass so this crement sequence is not a good choice. We can improve this sequence by adding 1 to increment if it is even.

In general it is better not to choose increments which are multiples of each other like 1,3,6,9 or 1,2,4,8 ecause the elements compared at one pass will be compared again at next pass. We can achieve greater fficiency if the values of increments are relatively prime. Many increment sequences have been suggested but l now none of them has been singled out as perfect. Knuth suggested a sequence

$=1$, $h_{i+1}= 3h_i+1$, stop at h_i when $h_i > (n-1)/9$.

or example if n=10000 then increments would be 1, 4, 13, 40, 121, 364, 1093, 3280.

Now we will see how we can implement shell sort. One method that naturally comes to mind is that in each ıss we sort all the sublists one by one using insertion sort. We will use another simple method which will ake our code look like insertion sort. We can just take all elements of the list one by one and insert each ement among the elements of its sublist. Here the elements will move h positions instead of one position.

```
/*P8.4 Program of sorting using shell sort*/
#include <stdio.h>
#define MAX 100
main()
{
    int arr[MAX],i,j,k,n,incr;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d",&arr[i]);
    }
```

```
        printf("\nEnter maximum increment (odd value) :. ");
        scanf("%d",&incr);

        /*Shell sort*/
        while(incr>=1)
        {
                for(i=incr; i<n; i++)
                {
                        k=arr[i];
                        for(j=i-incr; j>=0 && k<arr[j]; j=j-incr)
                                arr[j+incr]=arr[j];
                        arr[j+incr]=k;
                }
                incr=incr-2; /*Decrease the increment*/
        }/*End of while*/
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
        printf("\n");
}/*End of main()*/
```

This program is similar to the program of insertion sort, except for a few changes.

8.11.1 Analysis of Shell Sort

The coding of Shell sort technique is simple but its analysis is quite difficult and no one has been able to analyze it mathematically. The only results that are available are based on empirical studies. The running time depends on the number of increments and their value. The empirical studies show that for a particular sequence of increments it is of $O(n(\log n)^2)$ and for another sequence it is $O(n^{1.25})$. Shell sort is not a stable sort. It is an in-place sort and space complexity is O(1).

8.12 Merge Sort

Merge sort was developed by Jon von Neumann in 1945 and it has O(n log n) performance in both worst and average case. The main task in merge sort is merging two sorted lists into a single sorted list, so let's first examine the process of merging two sorted lists.

If there are two sorted arrays, then process of combining these sorted arrays into another sorted array is called merging. A simple approach could be to place the second array after the first and then sort the whole by applying any sorting technique, but by doing this we are not utilizing the fact that both the arrays are sorted. Since both arrays are sorted, we can merge them efficiently by making only one pass through each array. Let us take two arrays `arr1` and `arr2` in sorted order, we will combine them into a third sorted array `arr3`.

arr1- 5 8 9 28 34 arr2- 4 22 25 30 33 40 42

We will take one element from each array, compare them and then take the smaller one in third array. This process will continue until the elements of one array are finished. Then the remaining elements of unfinished array are put in the third array. The whole process for merging is shown in figure 8.11. `arr3` is the merged array, `i`, `j` and `k` are variables used for subscripts of `arr1`, `arr2` and `arr3` respectively.

Initially `i`, `j` and `k` point to the beginning of the three arrays so `i=0`, `j=0`, `k=0`. The elements `arr1[i]` and `arr2[j]` are compared, and the smaller one is copied to the third array `arr3` at position `k`. The variable `k` is incremented and one of the variables `i` or `j` is incremented.

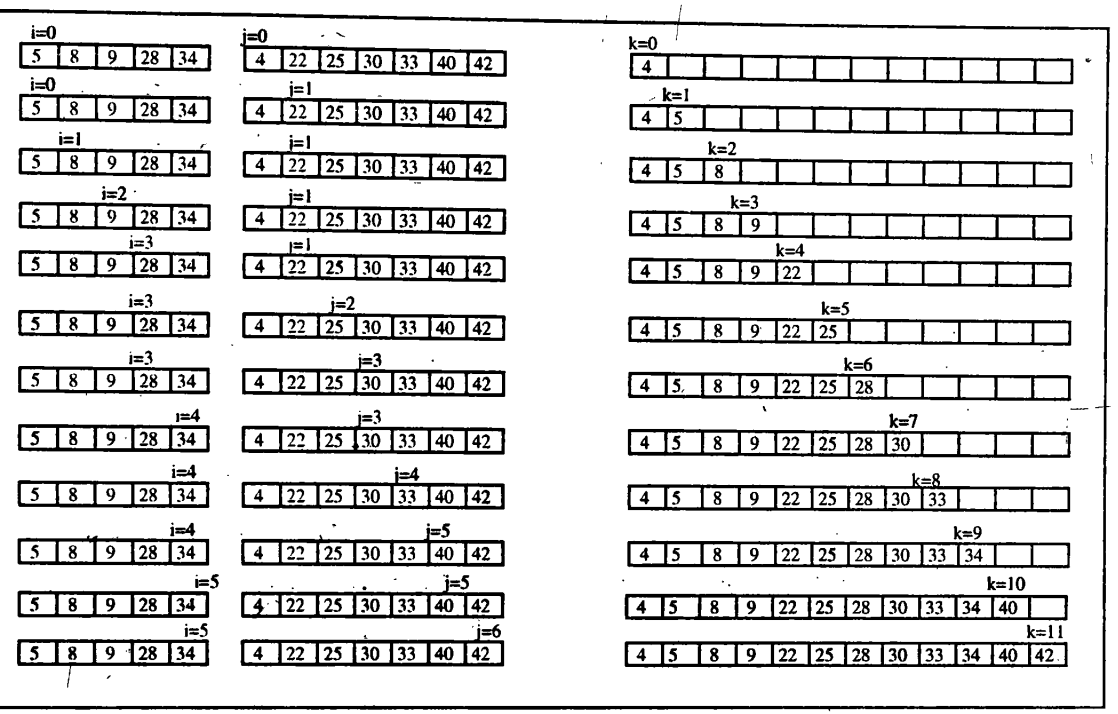


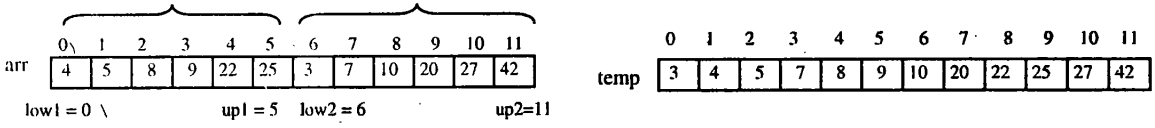
Figure 8.11 Merging

```
8.5 Program of merging two sorted arrays into a third sorted array*/
clude<stdio.h>
fine MAX 100
d merge(int arr1[],int arr2[],int arr3[],int n1,int n2);
n()

    int arr1[MAX],arr2[MAX],arr3[2*MAX],n1,n2,i;
    printf("Enter the number of elements in array 1 : ");
    scanf("%d",&n1);
    printf("Enter all the elements in sorted order :\n");
    for(i=0; i<n1; i++)
    {
            printf("Enter element %d : ",i+1);
            scanf("%d",&arr1[i]);
    }
    printf("Enter the number of elements in array 2 : ");
    scanf("%d",&n2);
    printf("Enter all the elements in sorted order :\n");
    for(i=0;i<n2;i++)
    {
            printf("Enter element %d : ",i+1);
            scanf("%d",&arr2[i]);
    }
    merge(arr1,arr2,arr3,n1,n2);
    printf("\nMerged list : ");
    for(i=0; i<n1+n2; i++)
            printf("%d ",arr3[i]);
    printf("\n");

d merge(int arr1[],int arr2[],int arr3[],int n1,int n2)

    int i,j,k;
```

```
        i = 0;  /*Index for first array*/
        j = 0;  /*Index for second array*/
        k = 0;  /*Index for merged array*/
        while((i <= n1-1 )&&(j <= n2-1))
        {
                if(arr1[i] < arr2[j])
                        arr3[k++] = arr1[i++];
                else
                        arr3[k++] = arr2[j++];
        }
        while(i <= n1-1) /*Put remaining elements of arr1 into arr3*/
                arr3[k++] = arr1[i++];
        while(j <= n2-1)        /*Put remaining elements of arr2 into arr3*/
                arr3[k++] = arr2[j++];
}/*End of merge()*/
```

Merging of two sorted lists of sizes n1 and n2 is O(n1+n2) since only one pass is made through each of th lists.

The two lists to be sorted need not be different arrays, they can be part of the same array. In the exampl given next, `arr[0:5]` and `arr[6:11]` are merged into another array `temp[0:11]`.



In the function merge() given next, we send the array `arr` and the lower and upper bounds of the lists to b merged. The array `temp` is also sent which will store the merged array.

```
merge(int arr[], int temp[],int low1, int up1, int low2, int up2)
{
        int i = low1, j = low2, k = low1;
        while(i<=up1 && j<=up2)
        {
                if(arr[i] <= arr[j])
                        temp[k++] = arr[i++];
                else
                        temp[k++] = arr[j++];
        }
        while(i<=up1)
                temp[k++] = arr[i++];
        while(j<=up2)
                temp[k++] = arr[j++];
}
```

If the total number of elements in arr is n, then the performance of the above merging algorithm O(n).

8.12.1 Top Down Merge Sort (Recursive)

This algorithm is based on the divide and conquer technique. The list is recursively divided till we get sing element lists which are obviously sorted, and then the lists are merged repeatedly to get a single sorted list. Th procedure for top down merge sort is-

1. Divide the list into two sublists of almost equal size.
2. Sort the left sublist recursively using merge sort.
3. Sort the right sublist recursively using merge sort.
4. Merge the two sorted sublists.

Terminating condition for recursion is when the sublist formed contains only one element. If the list contain odd number of elements then we assume that the left half is bigger. Let us take a list of numbers and sort it through merge sort. List is - 8 5 89 30 42 92 64 4 21 56 3

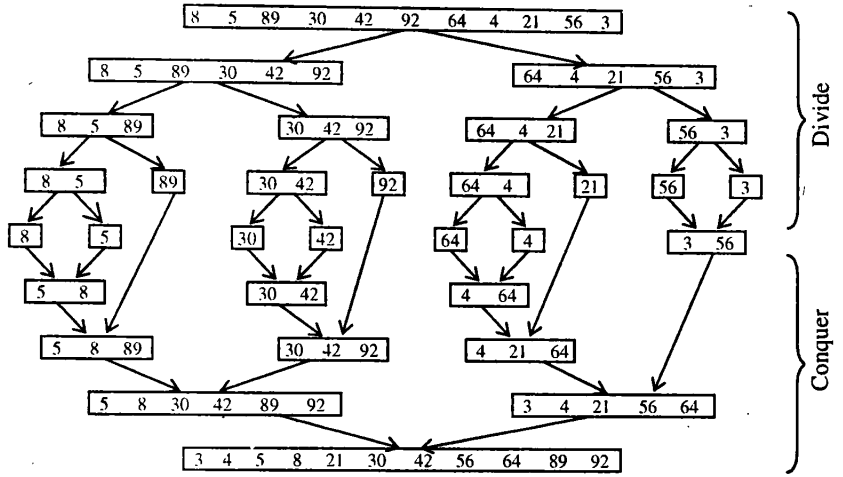


Figure 8.12 Top Down Merge Sort

- arr[0:10] is divided into arr[0:5] and arr[6:10]
  (8, 5, 89, 30, 42, 92, 64, 4, 21, 56, 3) -> (8, 5, 89, 30, 42, 92) and (64, 4, 21, 56, 3)
  - arr[0:5] is divided into arr[0:2] and arr[3:5]
    (8, 5, 89, 30, 42, 92) -> (8, 5, 89) and (30, 42, 92)
    - arr[0:2] is divided into arr[0:1] and arr[2]
      (8, 5, 89) -> (8, 5) and (89)
      - arr[0:1] is divided into arr[0] and arr[1]
        (8, 5) -> (8) and (5)
      - arr[0] and arr[1] are merged to produce sorted arr[0:1]
        (8) and (5) -> (5, 8)
    - arr[0:1] and arr[2] are merged to produce sorted arr[0:2]
      (5, 8) and (89) -> (5, 8, 89)
    - arr[3:5] is divided into arr[3:4] and arr[5]
      (30, 42, 92) -> (30, 42) and (92)
      - arr[3:4] is divided into arr[3] and arr[4]
        (30, 42) -> (30) and (42)
      - arr[3] and arr[4] are merged to produce sorted arr[3:4]
        (30) and (42) -> (30, 42)
    - arr[3:4] and arr[5] are merged to give sorted arr[3:5]
      (30, 42) and (92) -> (30, 42, 92)
  - arr[0:2] and arr[3:5] are merged to produce sorted arr[0:5]
    (5, 8, 89) and (30, 42, 92) -> (5, 8, 30, 42, 89, 92)
  - arr[6:10] is divided into arr[6:8] and arr[9:10]
    (64, 4, 21, 56, 3) -> (64, 4, 21) and (56, 3)
    - arr[6:8] is divided into arr[6:7] and arr[8]
      (64, 4, 21) -> (64, 4) and (21)
      - arr[6:7] is divided into arr[6] and arr[7]
        (64, 4) -> (64) and (4)
      - arr[6] and arr[7] are merged to produce sorted arr[6:7]
        (64) and (4) -> (4, 64)
    - arr[6:7] and arr[8] are merged to produce sorted arr[6:8]
      (4, 64) and (21) -> (4, 21, 64)
    - arr[9:10] is divided into arr[9] and arr[10]
      (56, 3) -> (56) and (3)
    - arr[9] and arr[10] are merged to produce sorted arr[9:10]
      (56) and (3) -> (3, 56)
  - arr[6:8] and arr[9:10] are merged to produce sorted arr[6:10]
    (4, 21, 64) and (3, 56) -> (3, 4, 21, 56, 64)
- arr[0:5] and arr[6:10] are merged to produce sorted arr[0:10]
  (5, 8, 30, 42, 89, 92) and (3, 4, 21, 56, 64) -> (3, 4, 5,8, 21, 30, 42, 56, 64, 89, 92)

```
/*P8.6 Program of sorting using merge sort through recursion*/
#include<stdio.h>
#define MAX 100
void merge_sort(int arr[],int low,int up);
void merge(int arr[],int temp[],int low1,int up1,int low2,int up2);
void copy(int arr[],int temp[],int low,int up);
main()
{
        int i,n,arr[MAX];
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        merge_sort(arr,0,n-1);
        printf("\nSorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
}/*End of main()*/
void merge_sort(int arr[],int low,int up)
{
        int mid;
        int temp[MAX];
        if(low<up)   /*if more than one element*/
        {
                mid = (low+up)/2;
                merge_sort(arr,low,mid);   /*Sort arr[low:mid]*/
                merge_sort(arr,mid+1,up);   /*Sort arr[mid+1:up]*/
                /*Merge arr[low:mid] and arr[mid+1:up] to temp[low:up]*/
                merge(arr,temp,low,mid,mid+1,up);
                copy(arr,temp,low,up); /* Copy temp[low:up] to arr[low:up] */

        }
}/*End of merge_sort*/
/*Merges arr[low1:up1] and arr[low2:up2] to temp[low1:up2]*/
void merge(int arr[],int temp[],int low1,int up1,int low2,int up2)
{
        int i = low1;
        int j = low2 ;
        int k = low1 ;
        while((i<=up1) && (j<=up2))
        {
                if(arr[i] <= arr[j])
                        temp[k++] = arr[i++] ;
                else
                        temp[k++] = arr[j++] ;
        }
        while(i<=up1)
                temp[k++] = arr[i++];
        while(j<=up2)
                temp[k++] = arr[j++];
}/*End of merge()*/
void copy(int arr[],int temp[],int low,int up)
{
        int i;
        for(i=low; i<=up; i++)
                arr[i] = temp[i];
}
```

The process is recursive so the record of the lower and upper bound of the sublists is implicitly maintained. When `merge_sort()` is called for the first time, the `low` and `up` are set to 0 and `n-1`. The value of `mid` is

lculated which is the index of middle element. Now the `merge_sort()` is called for left sublist which is `r[0:mid]`. The `merge_sort()` is recursively called for left sublists till it is called for a one element sub-list. In this call, value of `low` will not be less than `up`, it will be equal to `up` so recursion will terminate. Now `erge_sort()` is called for the right sublist of the previous recursive call. After the return of this call, the two blists are merged. This continues till the first recursive call returns after which we get the sorted result. Before turning from `merge_sort()`, the merged list which is in `temp` is copied back to `arr`. The following figure ows the values of variables `low` and `up` in the recursive calls of the function `merge_sort()`.

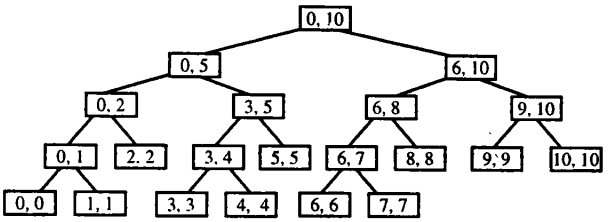


Figure 8.13

uick sort also uses divide and conquer approach but there partition step is difficult while combining is trivial, re in merge sort partition is simple but combining is difficult.

12.2 Analysis of Merge Sort

e know that n elements can be repeatedly divided into half approximately $\log_2 n$ times, so after halving the list $_2 n$ times we get n sublists of size 1. In each pass there will be merging of n elements which is O(n) so rformance of merge sort is $O(n \log_2 n)$.

Merge sort is a stable sort. Since the operation of merging is not in place, merge sort is also not an in place rt and requires O(n) extra space.

12.3 Bottom Up Merge Sort(Iterative)

e stack space needed for recursion can be avoided by implementing a non recursive version of merge sort. e iterative version works in bottom up manner and uses combine and conquer strategy.

this approach, the whole list is initially considered as n sorted sublists of size 1. The adjacent sublists of size are merged pairwise and we get n/2 sublists of size 2 (and possibly one more sublist of size 1 if n is odd). ese sublists of size 2 are then merged and we get n/4 sublists of size 4 (and possibly one sublist of size 1, 2 or This process continues until only one sublist of size n is left.

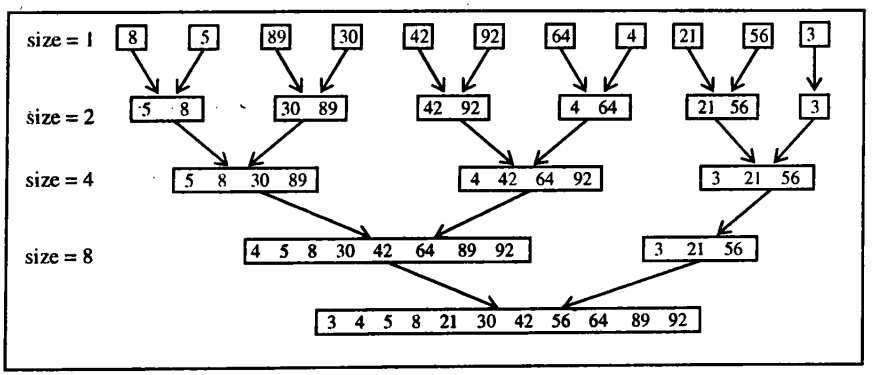


Figure 8.14 Bottom Up Merge Sort

If there is a sublist left in the last which cannot be merged with any sublist, it is just copied to the result. In the example given in figure 8.14, this case occurs in pass 1 where sublist [3] is left alone and in pass 3 where sublist [3, 21, 56] is left alone.

```
/*P8.7 Program of sorting using merge sort without recursion*/
#include<stdio.h>
#define MAX 100
void merge_sort(int arr[],int n);
void merge_pass(int arr[],int temp[],int size,int n);
void merge(int arr[],int temp[],int low1,int up1,int low2,int up2);
void copy(int arr[],int temp[],int n);
main()
{
        int arr[MAX],i,n;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        merge_sort(arr,n);
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
}/*End of main()*/
void merge_sort(int arr[],int n)
{
        int temp[MAX];
        int size = 1;
        while(size<n)
        {
                merge_pass(arr,temp,size,n);
                size = size*2;
        }
}
void merge_pass(int arr[],int temp[],int size,int n)
{
        int i,low1,up1,low2,up2;
        low1 = 0;
        while(low1+size < n)
        {
                up1 = low1 + size-1;
                low2 = low1 + size;
                up2 = low2 + size-1;
                if(up2 >= n)/*if length of last sublist is less than size*/
                up2 = n-1;
                merge(arr,temp,low1,up1,low2,up2);
                low1 = up2+1;  /*Take next two sublists for merging*/
        }
        for(i=low1; i<=n-1; i++)
                temp[i] = arr[i];     /*If any sublist is left*/
        copy(arr,temp,n);
}
void merge(int arr[],int temp[],int low1,int up1,int low2,int up2)
{
        int i = low1;
        int j = low2;
        int k = low1;
        while(i<=up1 && j<=up2)
        {
```

```
            if(arr[i] <= arr[j])
                    temp[k++] = arr[i++];
            else
                    temp[k++] = arr[j++];
    }
    while(i<=up1)
            temp[k++] = arr[i++];
    while(j<=up2)
            temp[k++] = arr[j++];

oid copy(int arr[], int temp[], int n)

    int i;
    for(i=0;i<n;i++)
            arr[i] = temp[i];
```

We can avoid the copying from arr to temp by merging alternately from arr to temp and from temp to rr. The function merge_sort() would be like this-

```
erge_sort(int arr[], int n)

    int temp[MAX], size = 1;;
    while(size<n)
    {
            merge_pass(arr,temp,size,n);
            size =size*2;
            merge_pass(temp,arr,size,n);
            size = size*2;
    }
```

Now we can delete the last statement from the function merge_pass() which is used for copying temp to rr. The total number of passes required is $\log_2 n$ and in each pass n elements are merged, so complexity is $O(n\log_2 n)$.

3.12.4 Merge Sort for linked List

When merge sort is used for linked lists, there is no need of temporary storage because merge operation in nked lists can be performed without temporary storage. Another advantage with linked list version is that there s no data movement, only the pointers are rearranged.

```
*P8.8 Program of merge sort for linked lists*/
include<stdio.h>
include<stdlib.h>
truct node

    int info;
    struct node *link;
;
truct node *merge_sort(struct node *p);
truct node *divide(struct node *p);
truct node *merge(struct node *p,struct node *q);
oid display(struct node *start);
truct node *create_list(struct node *start);
truct node *addatbeg(struct node *start,int data);
truct node *addatend(struct node *start,int data);
ain()

    struct node *start = NULL;
    start = create_list(start);
    start = merge_sort(start);
    display(start);
```

```
}
struct node *merge_sort(struct node *start)
{
        struct node *start_first, *start_second, *start_merged;
        if(start!=NULL && start->link!=NULL) /*if more than one element*/
        {
                start_first = start;
                start_second = divide(start);
                start_first = merge_sort(start_first);
                start_second = merge_sort(start_second);
                start_merged = merge(start_first,start_second);
                return start_merged;
        }
        else
                return start;
}
struct node *divide(struct node *p)
{
        struct node *q,*start_second;
        q = p->link->link;
        while(q!=NULL)
        {
                p = p->link;
                q = q->link;
                if(q!=NULL)
                        q = q->link;
        }
        start_second = p->link;
        p->link = NULL;
        return start_second;
}
struct node *merge(struct node *p1,struct node *p2)
{
        struct node *start_merged,*q;
        if(p1->info <= p2->info)
        {
                start_merged = p1;
                p1 = p1->link;
        }
        else
        {
                start_merged = p2;
                p2 = p2->link;
        }
        q = start_merged;
        while(p1!=NULL && p2!=NULL)
        {
                if(p1->info <= p2->info)
                {
                        q->link = p1;
                        q = q->link;
                        p1 = p1->link;
                }
                else
                {
                        q->link = p2;
                        q = q->link;
                        p2 = p2->link;
                }
        }
        if(p1!=NULL)
                q->link = p1;
```

```
        else
                q->link = p2;
        return start_merged;
}
void display(struct node *start)
{
        struct node *p;
        if(start==NULL)
        {
                printf("List is empty\n");
                return;
        }
        p = start;
        printf("List is :\n");
        while(p!=NULL)
        {
                printf("%d ",p->info);
                p = p->link;
        }
        printf("\n");
}/*End of display()*/
struct node *create_list(struct node *start)
{
        int i,n,data;
        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        start=NULL;
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        start=addatbeg(start,data);
        for(i=2;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                start=addatend(start,data);
        }
        return start;
}/*End of create_list()*/
struct node *addatbeg(struct node *start,int data)
{
        struct node *tmp;
        tmp = malloc(sizeof(struct node));
        tmp->info = data;
        tmp->link = start;
        start = tmp;
        return start;
}/*End of addatbeg()*/
struct node *addatend(struct node *start,int data)
{
        struct node *p,*tmp;
        tmp = malloc(sizeof(struct node));
        tmp->info = data;
        p = start;
        while(p->link!=NULL)
                p = p->link;
        p->link = tmp;
        tmp->link = NULL;
        return start;
}/*End of addatend()*/
```

The function `divide()` takes a pointer to the original list and returns a pointer to the start of the second sublist. We have taken two pointers `p` and `q`, where pointer `p` points to the first node and pointer `q` points to the third node. In a loop we move pointer `p` once and pointer `q` twice so that when the pointer `q` is `NULL`, pointer `p` points to the middle node. The node next to the middle node will be the start of second sublist. The middle node will be the last node of first sublist so its link is assigned `NULL`.

The function `merge()` takes pointers to the two sorted lists, merges them into a single sorted list and returns a pointer to the merged list. There is no requirement of any temporary storage for merging of linked lists.

8.12.5 Natural Merge Sort

Merge sort does not consider any sorted sequences inside a given list and sorts in usual manner. This type of merge sort is called straight merge sort, and in this sort all the sublists in a pass are of the same size except possibly the last one. Another type of merge sort is natural merge sort which takes advantage of the presence of sorted sequences(runs) in the list. For example consider this list-

12 45 67 3 66 21 89 90 98 65 43 56 68 96 23 87

The runs present in this list are (12, 45, 67) (3, 66) (21, 89, 90, 98) (65) (43, 56, 68, 96) (23, 87). In the first pass the runs are determined and these runs are merged instead of merging sublists of size 1, remaining passes are same as in the merge sort.

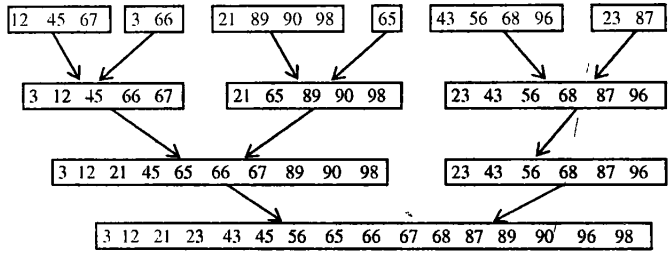


Figure 8.14 Natural Merge Sort

The disadvantage with this approach is that the size of sublists is different and we have to keep track of the start and end of each sublist. So this technique is beneficial only if there are many runs present in the data. Merge sort is not suitable for main memory sorts because it requires extra space of O(n) but this algorithm forms the basis of external sorting techniques. Merge sort is stable because the merge process is stable.

8.13 Quick Sort (Partition Exchange Sort)

Quick Sort was given by C.A.R. Hoare in 1962 and is based on the divide and conquer technique. In this technique, a problem is divided into small problems which are again divided into smaller problems and so on, hence solving the bigger problem reduces to solving these smaller ones. As the name suggests, quick sort is a very fast sorting technique and its average case performance is O(n log n). It is an in-place sort so no additional space is required for sorting. It is also known as partition exchange sort and is very efficient because the exchanges occur between elements that are far apart, and so less exchanges are needed to place an element in its final position.

Now let us see how this sorting is performed. We choose an element from the list and place it at its proper position in the list, i.e. at the position where it would be in the final sorted list. We call this element as pivot and it will be at its proper place if-

(i) all the elements to the left of pivot are less than or equal to the pivot and

(ii) all the elements to the right of pivot are greater than or equal to the pivot.

Any element of the list can be taken as pivot but for convenience the first element is taken as the pivot, we will talk more about the choice of pivot later. Suppose our list is [4, 6, 1, 8, 3, 9, 2, 7], if we take 4 as the pivot

then after placing 4 at its proper place the list becomes [1, 3, 2, **4**, 6, 8, 9, 7]. Now we can partition this list into two sublists based on this pivot and these sublists are [1, 3, 2] and [6, 8, 9, 7]. We can apply the same procedure to these two sublists separately i.e. we will select one pivot for each sublist and both the sublists are divided into 2 sublists each so now we get 4 sublists. This process is repeated for all the sublists that contain two or more elements and in the end we get our sorted list.

Now let us outline the process for sorting the elements through quick sort. Suppose we have an array `arr` with `low` and `up` as the lower and upper bounds.

1. Take the first element of list as pivot.
2. The list is partitioned in such a way that pivot comes at its proper place. After this partition, all elements to the left of pivot are less than or equal to the pivot and all elements to the right of pivot are greater than equal to the pivot. So one element of the list i.e. pivot is at its proper place. Let the index of pivot be `pivotloc`.
3. Create two sublists left and right side of pivot, left sublist is `arr[low].....arr[pivotloc-1]` and the right sublist is `arr[pivotloc+1]....arr[up]`.
4. The left sublist is sorted using quick sort recursively.
5. The right sublist is sorted using quick sort recursively.
6. The terminating condition for recursion is - when the sublist formed contains only one element or no element.

The sublists are kept in the same array, and there is no need of combining the sorted sublists at the end. Let us take a list of elements and sort them through quick sort.

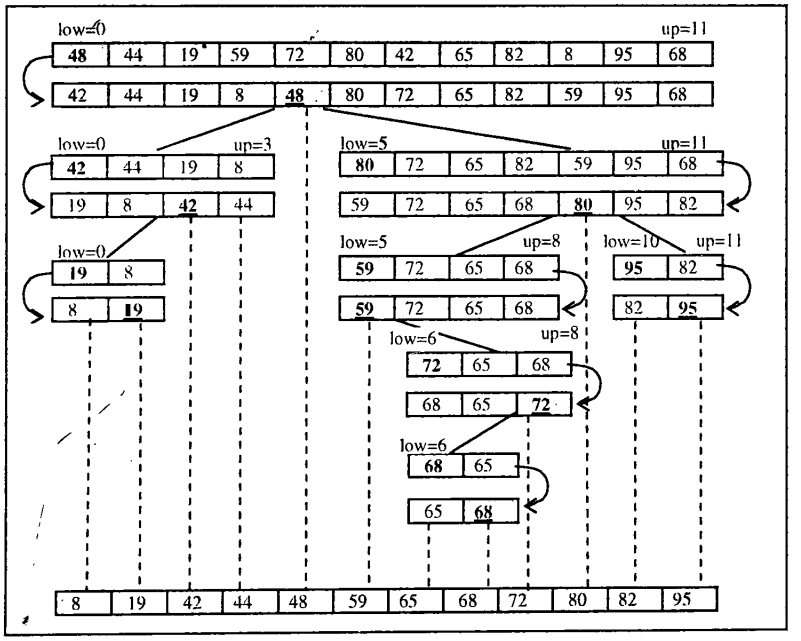


Figure 8.15 Quick Sort

Here we are focusing only on the recursion procedure; the logic of placing the pivot at proper place is discussed later. The values of `low` and `up` indicate the lower and upper bounds of the sublists.

Initially the list is [48, 44, 19, 59, 72, 80, 42, 65, 82, 8, 95, 68]. We will take the first element(48) as the pivot and place it in proper place and so the list becomes [42, 44, 19, 8, **48**, 80, 72, 65, 82, 59, 95, 68]. Now all the elements to the left of 48 are less than it and all elements to the right of 48 are greater than it. We will take two sublists left and right of 48 which are [42, 44, 19, 8] and [80, 72, 65, 82, 59, 95, 68], and sort them separately using the same procedure. Note that the order of the elements in the left sublist or in the right sublist is not the same as it appears in the original list. It depends on the partition process which is used to place pivot

at proper place. For now you just need to understand that all elements to left of pivot are less than or equal to pivot and all elements right of pivot are greater than or equal to pivot.

The left sublist is [42, 44, 19, 8] and its pivot is taken as 42, after placing the pivot the list becomes [19, 8, **42**, 44]. Now 42 is at its proper place, we again divide this list into two sublists which are [19, 8] and [44]. The list [44] has only one element so we will stop. The list [19, 8] is taken and here the pivot will be 19 and after placing the pivot at proper place the list becomes [8, **19**]. The left sublist is [8] and it contains only one element so we will not process it. There are no elements to the right of 19, so right sublist is not formed or we can say that right sublist contains zero elements so we will stop.

The right sublist is [80, 72, 65, 82, 59, 95, 68] and 80 is taken as the pivot. After placing the pivot the list becomes [59, 72, 65, 68, **80**, 95, 82]. Now the two sublists formed are [59, 72, 65, 68] and [95, 82]. In the first sublist, 59 is taken as the pivot and after placing it at the proper place the sublist becomes [**59**, 72, 65, 68]. There are no elements to the left of 59 so only one sublist is formed which is [72, 65, 68]. Here 72 is taken as the pivot and after placing it the list becomes [68, 65, **72**]. There are no elements to the right of 72 so only one sublist is formed which is [68, 65]. In this sublist 68 is taken as the pivot and after placing it at proper place the sublist becomes [65, **68**]. There are no elements to the right of 68 and the left sublist has only one element so we will stop.

After this we will take the sublist [95, 82]. Here 95 is taken as the pivot and after placing it at proper place the sublist is [82, **95**]. There are no elements to the right of 95 and the left sublist has only one element so we will stop.

We can write a recursive function for this procedure. There would be two terminating conditions, when the sublist formed has only 1 element or when no sublist is formed. If the value of `low` is equal to `up` then there will be only one element in the sublist and if the value of `low` exceeds `up` then no sublist is formed. So the terminating condition can be written as-

```
if(low>=up)
        return;
```

The recursive function `quick()` can be written as-

```
void quick(int arr[],int low,int up)
{
        int pivloc;
        if(low>=up)
                return;
        pivloc = partition(arr,low,up);
        quick(arr,low,pivloc-1); /*process left sublist*/
        quick(arr,pivloc+1,up);  /*process right sublist*/
}/*End of quick()*/
```

The function `partition()` will place the pivot at proper place and then return the location of pivot so that we can form two sublists left and right of pivot.

Since the process is recursive, the record of lower and upper bounds of sublists is implicitly maintained.

Now the second main task is to partition a list into two sublists by placing the pivot at the proper place. Let us see how we can do this. Suppose we have an array `arr[low:up]`, the element `arr[low]` will be taken as the pivot. We will take two index variables `i` and `j` where `i` is initialized to `low+1` and `j` is initialized to `up`. The following process will put the pivot at its proper place.

(a) Compare the pivot with `arr[i]`, and increment `i` if `arr[i]` is less than the pivot element. So the index variable `i` moves from left to right and stops when we get an element greater than or equal to the pivot.

(b) Now compare the pivot with `arr[j]`, and decrement `j` if `arr[j]` is greater than the pivot element. So the index variable `j` moves from right to left, and stops when we get an element less than or equal to the pivot.

(c) If `i` is less than `j`

Exchange the value of `arr[i]` and `arr[j]`, increment `i` and decrement `j`.

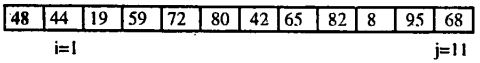
else

No exchange, increment `i`.

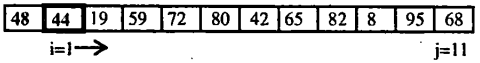
(d) Repeat the steps (a), (b), (c) till the value of `i` is less than or equal to `j`. We will stop when `i` exceeds `j`.
(e) When value of `i` becomes more than `j`, we have found proper place for pivot which is given by `j`, hence now pivot is to be placed at position `j`. Pivot was at location `low`, so we can place it at `j` by exchanging the value of `arr[low]` and `arr[j]`. Now pivot is at position `j` which is its final position.

Now let us take a list and see how pivot will be placed at proper place through this partition process.

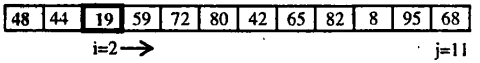
low=0, up=11
i=1 and j=11, Pivot = arr[low] = 48



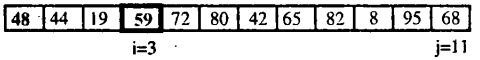
44 < 48, Increment i



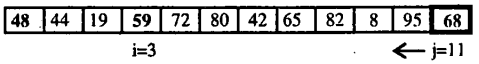
19 < 48, Increment i



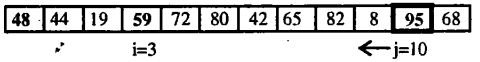
59 > 48, Stop i at 3



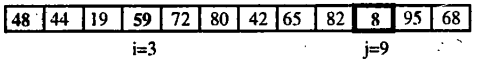
68 > 48, Decrement j



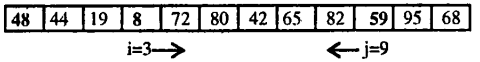
95 > 48, Decrement j



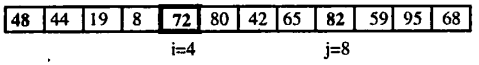
8 < 48, Stop j at 9.
Now both i and j stopped
Since i < j, Exchange arr[3] and arr[9]



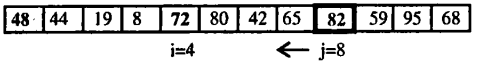
Increment i and decrement j



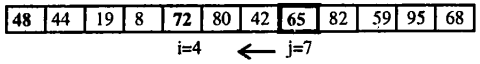
72 > 48, stop i at 4



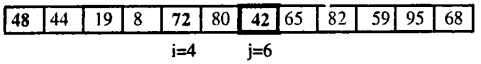
82 > 48, decrement j



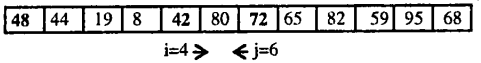
65 > 48, decrement j



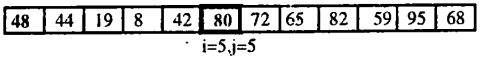
42 < 48, stop j at 6
Both i and j stopped
Since i < j, Exchange arr[4] and arr[6]



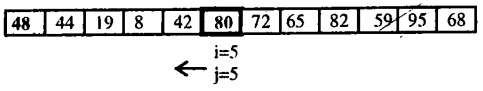
Increment i and decrement j



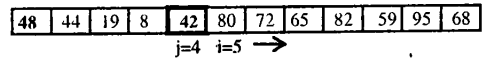
80 > 48, stop i at 5



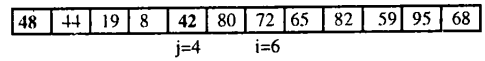
80 > 48, decrement j



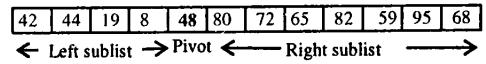
42 < 48, stop j at 4
Both i and j stopped.
i is not less than j, so no exchange
i is incremented



Now i > j, so stop movement of i and j.
j is the location for pivot.
Exchange arr[0] and arr[4]



Pivot placed at proper place



The location for pivot is the value of `j`, so the function partition will return value of `j`. Now for left sublist `low=0` and `up=3` and for right sublist `low=5` and `up=11`.

```
int partition(int arr[],int low,int up)
{
        int temp,i,j,pivot;
        i = low+1;
        j = up;
        pivot = arr[low];
        while(i <= j)
        {
                while((arr[i] < pivot) && (i < up))
                        i++;
                while(arr[j] > pivot)
                        j--;
                if(i < j)
                {
                        temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                        i++;
                        j--;
                }
                else
                        i++;
        }
        arr[low] = arr[j];
        arr[j] = pivot;
        return j;
}/*End of partition()*/
```

The variable `i` is moving right, so we have to prevent it from moving past the array bound. For example if pivot is the largest element in the array then there is no element in the array that can stop `i`, and it just moves on. So we have to check the condition `(i<up)` before incrementing `i`. The other way out may be to put a sentinel at the end having largest possible value. The variable `j` is moving left but it will never cross the bound of the array because pivot is there at the leftmost position.

The program for quick sort is given below, the functions `quick()` and `partition()` are the same as described earlier.

```
/*P8.9 Program of sorting using quick sort*/
#include<stdio.h>
#define MAX 100
void quick(int arr[],int low,int up);
int partition(int arr[],int low,int up);
main()
{
        int array[MAX],n,i;
```

```
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
              printf("Enter element %d : ",i+1);
              scanf("%d",&array[i]);
        }
        quick(array,0,n-1);
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
              printf("%d ",array[i]);
        printf("\n");
}/*End of main()*/
```

The partition process is not stable so quick sort is not a stable sort.

8.13.1 Analysis of Quick Sort

The time requirement of quick sort depends on the relative size of the two sublists formed. If the partition is balanced and the two sublists are of almost equal size then the sorting is fast while if the partition is unbalanced and one sublist is much larger than the other, then the sorting is slow.

The best case for quick sort would be when the pivot is always placed in the middle of the list, i.e. when we get two sublists of equal size. If we have a list of n elements, then we get 2 sublists of size approximately n/2 which are again divided equally so that we get 4 sublists of size approximately n/4 each and these are again divided and we get 8 sublists of size approximately n/8 each and so on. We know that n elements can be repeatedly divided into half approximately $\log_2 n$ times, so after halving the list $\log_2 n$ times we get n sublists of size 1.

Now let's find out the number of comparisons that will be performed. Initially we have 1 list of size n for which there will be approximately n comparisons (exactly n-1, but we want Oh notation only so we can take n), then we have 2 lists of size approximately n/2 each so there will be approximately 2*(n/2) comparisons, then we have 4 lists of size approximately n/4 each so there will be approximately 4*(n/4) comparisons and so on. So the total number of comparisons would be approximately equal to-

$n + 2*(n/2) + 4*(n/4) + 8*(n/8) + \ldots\ldots\ldots\ldots n*(n/n)$

$n + n + n + \ldots\ldots\ldots\ldots + n$ ($\log_2 n$ terms)

$= n\log_2 n$

So in best case the performance of quick sort is $O(n\log_2 n)$. Now let us see how quick sort performs in worst case. If the pivot is the smallest or largest element of the list, then it divides the list into two sublists from which one is empty and other contains n-1 elements. The worst case occurs if this situation arises in every recursive call. If the first element is taken as the pivot then this worst case occurs when the list is fully sorted(or reverse sorted).

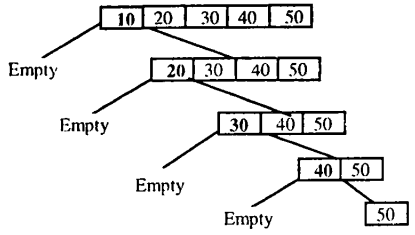


Figure 8.16

If we have a list of n elements, then first we get 2 sublists of sizes 0 and n-1. Now sublist of size n-1 is divided into two sublists of sizes 0 and n-2. The total number of lists that are sorted is n-1 and these are of sizes n, n-1, n-2, ...2. The total number of comparisons will be-

$n-1 + n-2 + n-3 + ... + 1$

$= n(n-1)/2$

$= O(n^2)$

So in worst case the performance of quick sort is $O(n^2)$.

The average case performance is closer to the best case than to the worst case and is found out to be $O(n\log_2 n)$. It is not a stable sort. Space complexity for this sort is O(log n).

8.13.2 Choice of pivot in Quick Sort

The quick sort performs best when the pivot is chosen such that it divides the list into two equal sublists. It performs badly if the pivot value does an unbalanced partition i.e. one sublist is very small and the other one is big. So the efficiency of quick sort can be improved by choosing a pivot value which makes balanced partitions. The worst case can be avoided by careful selection of the pivot.

The original algorithm given by Hoare selected the first element as the pivot but the first element is not a good choice because if the list is sorted or almost sorted then partitions will be unbalanced and performance will not be good. The last element is also not a good choice for similar reasons. As the chances of a list being sorted or almost sorted are high, it is better to avoid the first and last elements.

Another option is to choose the pivot randomly. For this we chose a random number `k` between `low` and `up`, and take `arr[k]` as the pivot. This `arr[k]` is interchanged with the first element before the while loop in the function `partition()` and rest of the code remains same. It seems to be a safe option but random number generators are time consuming.

The ideal choice would be to take the median value of all the elements but this option is costly so we can use median of three method. In this method we take the median of the first, middle and last elements i.e. median of { `arr[low]`, `arr[(low+up)/2]`, `arr[up]` } and then interchange the median element with the first element.

```
mid = (low+up)/2;
if(arr[low] > arr[mid])
        exchange(arr[low],arr[mid]);
if(arr[low] > arr[up])
        exchange(arr[low],arr[up]);
if(arr[mid] > arr[right])
        exchange(arr[mid],arr[right]);
exchange(arr[low],arr[mid]);
```

In the process of finding the median we have placed the largest of three at the end, so pivot cannot be greater than the last element and hence now there is no need of condition `i<up` before incrementing `i`.

8.13.3 Duplicate elements in quick sort

In our program we stop variables `i` and `j` when we find an element equal to the pivot. Let us see what other options are and why they are less efficient. There can be 4 options when an element equal to pivot is encountered

(i) stop `i` and move `j`

(ii) stop `j` and move `i`

(iii) stop both `i` and `j`

(iv) move both `i` and `j`

If we stop one pointer and move another then all elements equal to the pivot would go in one sublist and the partitioning will be unbalanced. For example if we stop `i` and move `j`, then all the equal elements go to the right sublist, and if we stop `j` and move `i` then all the equal elements go to the left sublist. The first two options are

not good because they tend to maximize the difference in the sizes of the sublists which reduces the efficiency of quick sort.

To understand which of the last two options is better, let us consider the case when all the elements in the list are equal. If both `i` and `j` stop then there will be many unnecessary exchanges between equal elements but the good thing is that `i` and `j` will meet somewhere in middle of the list thus creating two almost equal sublists. If both `i` and `j` move then no unnecessary swaps would be there but the sublists would be unbalanced. If all the elements are equal then `j` will always stop at the leftmost position and so pivot will always be placed at the leftmost position and hence one sublist will always be empty. This is the same situation that we studied in the worst case and the running time is $O(n^2)$.

So it is best to stop `i` and `j` when any element equal to the pivot is encountered. It might seem that considering the case of all equal elements is not a good idea as it would rarely occur. A list of all equal elements can be rare but we may get a sublist consisting of all equal elements, for example suppose we have a list of 500,000 elements out of which 10,000 are equal. Since quick sort is recursive so at some point there will be a recursive call for these 10,000 identical elements and if sorting these elements takes quadratic time then the overall efficiency will be affected.

8.14 Binary tree sort

Binary tree sort uses a binary search tree for sorting the data. This algorithm proceeds in two phases, namely construction phase and traversal phase.

1. Construction Phase - A binary search tree is created from the given data.
2. Traversal Phase - The binary search tree created is traversed in inorder to obtain the sorted data.

The details of creation and traversal of a binary search tree are discussed in chapter on trees. Let us take an array `arr` containing unsorted elements and sort them using binary tree sort.

19	35	10	12	46	6	40	3	90	8
0	1	2	3	4	5	6	7	8	9

First we will insert all the elements of this array in a binary search tree one by one.

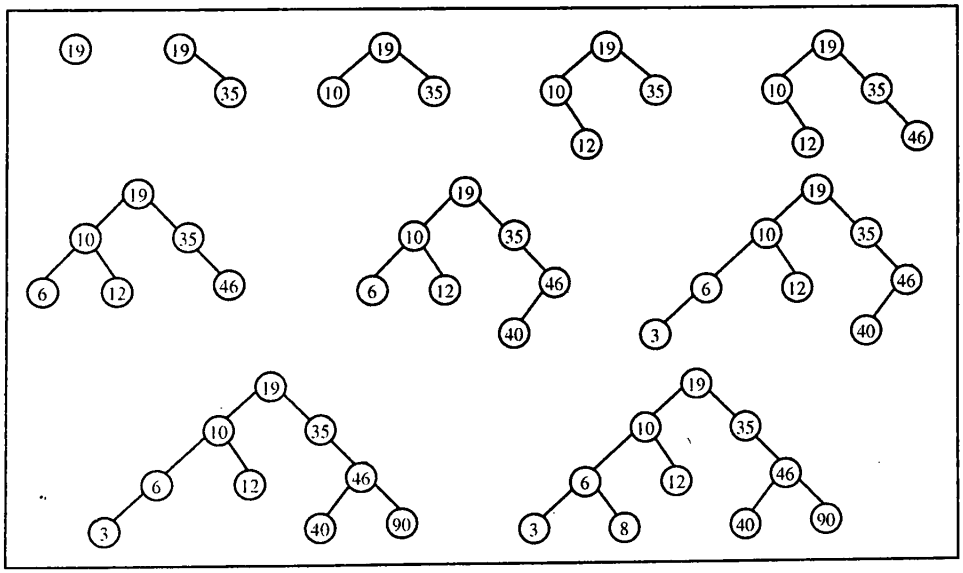


Figure 8.17 Binary Tree Sort

The inorder traversal of this binary search tree is-

3, 6, 8, 10, 12, 19, 35, 40, 46, 90

These elements are copied back to the array in this order and we get sorted array.

3	6	8	10	12	19	35	40	46	90
0	1	2	3	4	5	6	7	8	9

```
/*P8.10 Binary tree Sort*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
struct node
{
        struct node *lchild;
        int info;
        struct node *rchild;
};
struct node *stack[MAX];
int top=-1;
void push_stack(struct node *item);
struct node *pop_stack();
int stack_empty();
struct node *insert(struct node *ptr, int item);
void inorder(struct node *ptr, int arr[]);
struct node *Destroy(struct node *ptr);
main()
{
        struct node *root=NULL;
        int arr[MAX],n,i;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        for(i=0; i<n; i++)
                root = insert(root, arr[i]);
        inorder(root, arr);
        printf("\nSorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ", arr[i]);
        root=Destroy(root);
}/*End of main( )*/
struct node *insert(struct node *root, int ikey)
{
        struct node *tmp,*par,*ptr;
        ptr = root;
        par = NULL;
        while(ptr!=NULL)
        {
                par = ptr;
                if(ikey < ptr->info)
                        ptr = ptr->lchild;
                else
                        ptr = ptr->rchild;
        }
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=ikey;
        tmp->lchild=NULL;
        tmp->rchild=NULL;
        if(par==NULL)
                root=tmp;
        else if(ikey < par->info)
```

```
                par->lchild=tmp;
        else
                par->rchild=tmp;
        return root;
}/*End of insert( )*/
void inorder(struct node *root, int arr[])
{
        struct node *ptr=root;
        int i=0;
        if(ptr==NULL)
        {
                printf("Tree is empty\n");
                return;
        }
        while(1)
        {
                while(ptr->lchild!=NULL)
                {
                        push_stack(ptr);
                        ptr = ptr->lchild;
                }
                while(ptr->rchild==NULL)
                {
                        arr[i++]=ptr->info;
                        if(stack_empty())
                                return;
                        ptr = pop_stack();
                }
                arr[i++]=ptr->info;
                ptr = ptr->rchild;
        }
}/*End of inorder( )*/
/*Delete all nodes of the tree*/
struct node *Destroy(struct node *ptr)
{
        if(ptr!=NULL)
        {
                Destroy(ptr->lchild);
                Destroy(ptr->rchild);
                free(ptr);
        }
        return NULL;
}/*End of Destroy()*/
void push_stack(struct node *item)
{
        if(top==(MAX-1))
        {
                printf("Stack Overflow\n");
                return;
        }
        stack[++top]=item;
}/*End of push_stack()*/
struct node *pop_stack()
{
        struct node *item;
        if(top==-1)
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        item=stack[top--];
```

```
        return item;
}/*End of pop_stack()*/
int stack_empty()
{
        if(top==-1)
                return 1;
        else
                return 0;
} /*End of stack_empty*/
```

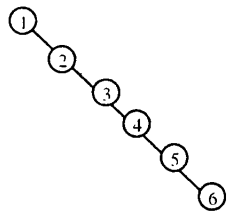
8.14.1 Analysis of Binary Tree Sort

We have seen that if a binary tree contains n nodes, then its maximum height possible is n and minimum height possible is $\lceil \log_2(n+1) \rceil$.

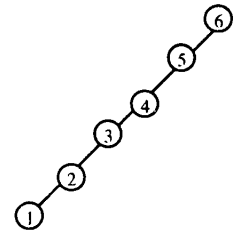
The maximum height n occurs when tree reduces to a chain or linear structure and in case of binary search tree this happens if the elements are inserted in ascending or descending order. The minimum height occurs when the tree is balanced.

The main operation in binary tree sort is the insertion of elements in binary search tree. We have studied if h is the height of a binary search tree, then all the basic operations run in order O(h). Insertion of an element is also O(h) and so insertion of n elements will be O(nh). If the data that is to be sorted is in ascending or descending order then the tree that we get by inserting this data will have height n and so the insertion of n elements will be $O(n^2)$. If the data gives us an almost balanced tree then the insertion of n elements will be O(n log n).

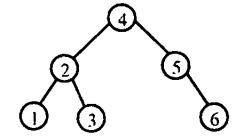
Let us find out this order by counting the number of comparisons. We will take 6 numbers 1, 2, 3, 4, 5, 6 in sorted order, reverse sorted order and random order. The binary search trees obtained by inserting these numbers in different orders are -



(a) Data in sorted order
{ 1, 2, 3, 4, 5, 6}
Total comparisons =
0 + 2+ 3 +4 + 5+ 6 = 20



(b) Data in reverse sorted order
{ 6, 5, 4, 3, 2, 1}
Total Comparisons =
0 + 2+ 3 + 4 + 5+6 = 20



(c) Data in random order
{ 4, 2, 1, 5, 3, 6}
Total comparisons =
0 + 2 + 2 + 3 + 3 + 3 = 13

Figure 8.18

In case (a), no comparison is needed to insert node 1, only 2 comparisons are needed to insert node 2, and 3 comparisons are needed to insert node 3, and so on. Hence the total number of comparisons required to insert all the nodes is 20. Similarly in case (b) no comparison is needed to insert node 6, only 2 comparisons are needed to insert node 5, and 3 comparisons are needed to insert node 4, and so on. Here also the total number of comparisons required to insert all the nodes is 20. In case (c) no comparison is needed to insert node 4, and 2 comparisons each are required to insert nodes 2 and 5, and 3 comparisons each are required for inserting nodes 1, 3, 6 and so total number of comparisons in this case is only 13.

If the data is in sorted order or reverse sorted order then the total number of comparisons is given by-

$0 + 2 + 3 + 4 + 5 + \ldots\ldots\ldots\ldots\ldots + n = n(n+1)/2 \Rightarrow O(n^2)$

The main drawback of the binary tree sort is that if data is already in sorted order or in reverse order then the performance of binary tree sort is not good.

If data is in random order and suppose we get a balanced tree whose height is approximately log n then the number of comparisons can be given by-

$0 + 2 * 2^1 + 3 * 2^2 + 4 * 2^3 + \ldots\ldots\ldots (h) * 2^{h-1}$

This is because there can be maximum 2^L nodes at any level L, and L+1 comparisons are required to insert any node at level L as we have seen in case (c). The root node is at level 0 and the last level is h-1. The efficiency in this case is O(n log n).

So if the tree that we obtain is of height log n then the performance of binary tree sort is O(n log n). Generally with random data the chances of getting a balanced tree are good so we can say that average run time of binary tree sort is O(n log n).

After insertion of elements, some time is needed for traversal also. If we use a threaded tree then we can reduce this time by avoiding the use of stack. Binary tree sort is not an in-place sort since it requires additional O(n) space for the construction of tree. It is a stable sort.

8.15 Heap Sort

The heap tree that was introduced in the chapter on Trees has an important application in sorting.
Heap sort is performed in two phases-

Phase 1 - Build a max heap from the given elements.
Phase 2 - Keep on deleting the root till there is only one element in the tree.

The root of a heap always has the largest element so by successively deleting the root, we get the elements in descending order i.e. first the largest element of list will be deleted then second largest and so on. We can store the deleted elements in a separate array or move them to the end of the same array that represents the heap.

If we have `n` elements that are to be sorted, first we build a heap of size `n`. The elements `arr[1], arr[2],.....arr[n]` form the heap. Then root is deleted and we get a heap of size `n-1`. So now the elements `arr[1],arr[2],.....arr[n-1]` form the heap but `arr[n]` is not a part of the heap. The element deleted from root is the largest element, we can store it in `arr[n]` because `arr[n]` is not a part of heap now. Now the root from heap of size `n-1` is deleted and we get a heap of size `n-2`. The element deleted from root can be stored in `arr[n-1]`. This process goes on till we delete the root from heap of size 2 and get a heap of size 1 and this time the element deleted from root is stored in `arr[2]`. This way the array that represented the heap becomes sorted.

Let us take an array and sort it by applying heap tree sort.

arr	25	35	18	9	46	70	48	23	78	12	95
	1	2	3	4	5	6	7	8	9	10	11

First this array is converted to a heap. The procedure of building a heap is described in chapter on trees. The heap obtained from this array is shown below-

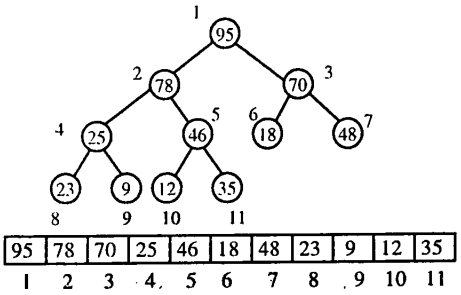
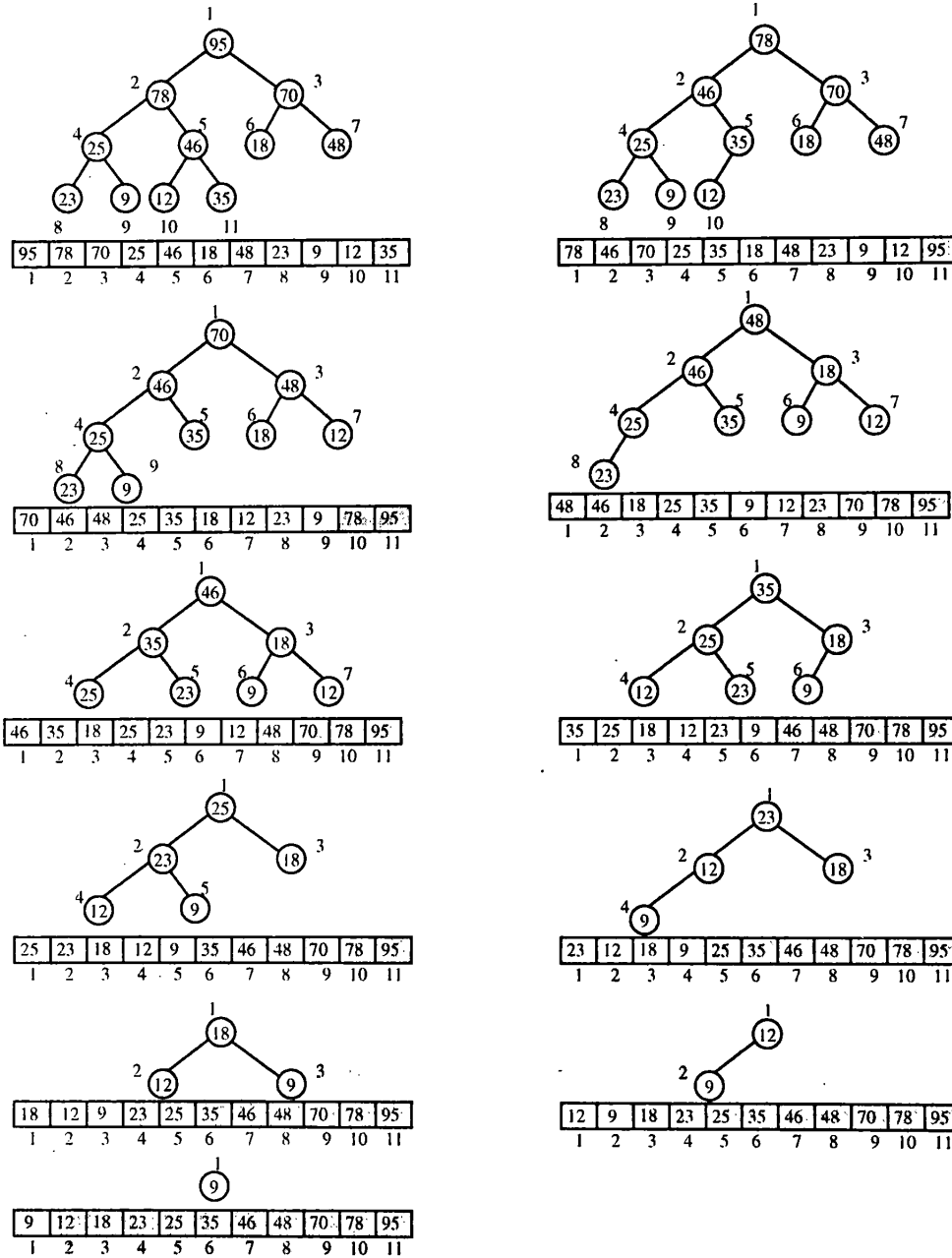


Figure 8.19

Now the root is repeatedly deleted from the heap. The procedure of deletion of root from the heap is described in chapter on trees.



This way finally we see that the array `arr` becomes sorted. If we don't want to change the array that represents the heap we can take a separate array `s_arr[]` for the sorted output. The elements deleted from root can be stored in `s_arr[n]`, `s_srr[n-1]` and so on till `arr[1]`. This time we have to copy the element in position 1 also.

```
/*P8.11 Program of sorting through heapsort*/
```

```
#include <stdio.h>
#define MAX 100
void heap_sort(int arr[],int size);
void buildHeap(int arr[],int size);
int del_root(int arr[],int *size);
void restoreDown(int arr[],int i,int size);
void display(int arr[],int n);
main()
{
        int i,n arr[MAX];
        printf("Enter number of elements : ");
        scanf("%d",&n);
        for(i=1; i<=n; i++)
        {
                printf("Enter element %d : ",i);
                scanf("%d",&arr[i]);
        }
        printf("Entered list is :\n");
        display(arr,n);
        heap_sort(arr,n);
        printf("Sorted list is :\n");
        display(arr,n);
}/*End of main()*/

void heap_sort(int arr[],int size)
{
        int max;
        buildHeap(arr,size);
        printf("Heap is : ");
        display(arr,size);
        while(size>1)
        {
                max = del_root(arr,&size);
                arr[size+1]=max;
        }
}/*End of heap_sort*/

void buildHeap(int arr[],int size)
{
        int i;
        for(i=size/2; i>=1; i--)
                restoreDown(arr,i,size);
}/*End of buildHeap()*/

int del_root(int arr[],int *size)
{
        int max = arr[1];
        arr[1] = arr[*size];
        (*size)--;
        restoreDown(arr,1,*size);
        return max;
}/*End of del_root()*/

void restoreDown(int arr[],int i,int size )
{
        int left=2*i, right=left+1;
        int num = arr[i];
        while(right<=size)
        {
                if(num>=arr[left] && num>=arr[right])
                {
                        arr[i] = num;
                        return;
                }
                else if(arr[left] > arr[right])
```

```
                {
                        arr[i] = arr[left];
                        i = left;
                }
                else
                {
                        arr[i] = arr[right];
                        i = right;
                }
                left = 2 * i;
                right = left + 1;
        }
        if(left == size && num < arr[left])  /*when right == size+1*/
        {
                arr[i] = arr[left];
                i = left;
        }
        arr[i] = num;
}/*End of restoreDown()*/
void display(int arr[],int n)
{
        int i;
        for(i=1;i<=n;i++)
                printf("%d  ",arr[i]);
        printf("\n");
}/*End of display()*/
```

8.15.1 Analysis of Heap Sort

The algorithm of heap sort proceeds in two phases. In the first phase we build a heap and its running time is O(n) if we use the bottom up approach. The delete operation in a heap takes O(log n) time, and it is called n-1 times, hence the complexity of second phase is O(nlog n). So the worst case complexity of heap sort is O(nlog n). The average case and best case complexity is also O(nlog n).

Heap sort is good for larger lists but it is not preferable for small list of data. It is not a stable sort. It has no need of extra space other than one temporary variable so it is an in place sort and space complexity is O(1).

8.16 Radix Sort

Radix sort is a very old sorting technique and was used to sort punch cards before the invention of computers. If we are given a list of some names and asked to sort them alphabetically, then we would intuitively proceed by first dividing the names into 26 piles with each pile containing names that start with the same alphabet. In the first pile we will put all the names which start with alphabet 'A', in the second pile we will put all the names which start with alphabet 'B' and so on. After this each pile is further divided into subpiles according to the second alphabet of the names. This process of creating subpiles will continue and the list will become totally sorted when the number of times subpiles are created is equal to the number of alphabets in the largest name.

In the above case, where we had to sort names the radix was 26(all alphabets). If we have a list of decimal numbers then the radix will be 10 (digits 0 to 9). We have sorted the names starting from the most significant position (from left to right). While sorting numbers also we can do the same thing. If all the numbers don't have same number of digits then we can add leading zeros. First all numbers are divided into 10 groups based on most significant digit, and then these groups are again divided into subgroups based on the next significant digit. The problem with the implementation of this method is that we have to keep track of lots of groups and subgroups. The implementation would be much simpler if the sorting starts from the least significant digit(from right to left). These two methods are called most significant digit (MSD) radix sort and least significant digit (LSD) radix sort.

In LSD radix sort, sorting is performed digit by digit starting from the least significant digit and ending at the most significant digit. In first pass, elements are sorted according to their units (least significant) digits, in second pass elements are sorted according to their tens digit, in third pass elements are sorted according to hundreds digit and so on till the last pass where elements are sorted according to their most significant digits.

For implementing this method we will take ten separate queues for each digit from 0 to 9. In the first pass, numbers are inserted into appropriate queues depending on their least significant digits(units digit), for example 283 will be inserted in queue 3 and 540 will be inserted in queue 0. After this all the queues are combined starting from the digit 0 queue to digit 9 queue and a single list is formed. Now in the second pass the numbers from this new list are inserted in queues based on tens digit, for example 283 will be inserted in queue 8 and 540 will be inserted in queue 4. These queues are combined to form a single list which is used in third pass. This process continues till numbers are inserted into queues based on the most significant digit. The single list that we will get after combining these queues will be the sorted list. We can see that the total number of passes will be equal to the number of digits in the largest number. Let us take some numbers in unsorted order and sort them by applying radix sort.

Original List : 62, 234, 456, 750, 789, 3, 21, 345, 983, 99, 153, 65, 23, 5, 98, 10, 6, 372

Pass 1 : Numbers classified according to units digit(least significant digit)

Queue for digit 0	750, 10
Queue for digit 1	21
Queue for digit 2	62, 372
Queue for digit 3	3, 983, 153, 23
Queue for digit 4	234
Queue for digit 5	345, 65, 5
Queue for digit 6	456, 6
Queue for digit 7	
Queue for digit 8	98
Queue for digit 9	789, 99

List after first pass : 750, 10, 21, 62, 372, 3, 983, 153, 23, 234, 345, 65, 5, 456, 6, 98, 789, 99

Pass 2 : Numbers classified according to tens digit

Queue for digit 0	3, 5, 6
Queue for digit 1	10
Queue for digit 2	21, 23
Queue for digit 3	234
Queue for digit 4	345
Queue for digit 5	750, 153, 456
Queue for digit 6	62, 65
Queue for digit 7	372
Queue for digit 8	983, 789
Queue for digit 9	98, 99

List after second pass : 3, 5, 6, 10, 21, 23, 234, 345, 750, 153, 456, 62, 65, 372, 983, 789, 98, 99

Pass 3 : Numbers classified according to hundreds digit(most significant digit)

Queue for digit 0	3, 5, 6, 10, 21, 23, 62, 65, 98, 99
Queue for digit 1	153
Queue for digit 2	234
Queue for digit 3	345, 372
Queue for digit 4	456
Queue for digit 5	
Queue for digit 6	
Queue for digit 7	750, 789
Queue for digit 8	
Queue for digit 9	983

List after third pass : 3, 5, 6, 10, 21, 23, 62, 65, 98, 99, 153, 234, 345, 372, 456, 750, 789, 983

Sorted List : 3, 5, 6, 10, 21, 23, 62, 65, 98, 99, 153, 234, 345, 372, 456, 750, 789, 983

Figure 8.20 Radix Sort

It is important that in each pass the sorting on digits should be stable i.e. numbers which have same i^{th} digit(from right) should remain sorted on the $(i-1)^{th}$ digit(from right). After any pass when we get a single list, we should enter the numbers in the queue in the same order as they are in list. This will ensure that the digit sorts are stable.

We take the initial input in linked list to simplify the process. If the input is in array then we can convert it to linked list. In the program we just traverse the linked list and add the number to the appropriate queue. After this we can combine the queues into one single list by joining the end of a queue to start of another queue.

We do not know in advance how many numbers will be inserted in a particular queue in any pass. It may be possible that in a particular pass, the digit is same for all the numbers, and all numbers may have to be inserted in the same queue. If we use arrays for implementing queues then each array should be of size n, and we will need space equal to 10*n. So it is better to take linked allocation of queues instead of sequential allocation.

```
/*P8.12 Sorting using radix sort*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
}*start = NULL;
void radix_sort();
int large_dig();
int digit(int number,int k);
main()
{
        struct node *tmp,*q;
        int i,n,item;
        printf("Enter the number of elements in the list : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&item);
                /*Inserting elements in the linked list*/
                tmp = malloc(sizeof(struct node));
                tmp->info = item;
                tmp->link = NULL;
                if(start==NULL) /*Inserting first element */
                        start = tmp;
                else
                {
                        q = start;
                        while(q->link!=NULL)
                                q = q->link;
                        q->link = tmp;
                }
        }/*End of for*/
        radix_sort();
        printf("Sorted list is :\n");
        q = start;
        while(q!=NULL)
        {
                printf("%d ",q->info);
                q = q->link;
        }
        printf("\n");
}/*End of main()*/
void radix_sort()
{
        int i,k,dig,least_sig,most_sig;
```

```
    struct node *p,*rear[10],*front[10];
    least_sig = 1;
    most_sig = large_dig(start);
    for(k=least_sig; k<=most_sig; k++)
    {
        /*Make all the queues empty at the beginning of each pass*/
        for(i=0; i<=9; i++)
        {
            rear[i] = NULL;
            front[i] = NULL ;
        }
        for(p=start; p!=NULL; p=p->link)
        {
            dig = digit(p->info,k); /*Find kth digit in the number*/
            /*Add the number to queue of dig*/
            if(front[dig] == NULL)
                front[dig] = p ;
            else
                rear[dig]->link = p;
            rear[dig] = p;
        }
        /*Join all the queues to form the new linked list*/
        i = 0;
        while(front[i] == NULL)
            i++;
        start = front[i];
        while(i<9)
        {
            if(rear[i+1]!=NULL)
                rear[i]->link = front[i+1];
            else
                rear[i+1] = rear[i];
            i++;
        }
        rear[9]->link = NULL;
    }
/*End of radix_sort*/
/*This function finds number of digits in the largest element of the list */
int large_dig()

    struct node *p=start;
    int large=0,ndig=0;
    /*Find largest element*/
    while(p!=NULL)
    {
        if(p->info > large)
            large = p->info;
        p = p->link ;
    }
    /*Find number of digits in largest element*/
    while(large!=0)
    {
        ndig++;
        large = large/10 ;
    }
    return(ndig);
/*End of large_dig()*/
/*This function returns kth digit of a number*/
int digit(int number,int k)

    int digit,i;
    for(i=1; i<=k; i++)
```

```
        {
                digit = number%10 ;
                number = number/10 ;
        }
        return(digit);
}/*End of digit()*/
```

8.16.1 Analysis of Radix Sort

The number of passes p is equal to the number of digits in largest element and in each pass we have n operations where n is the total number of elements. In the program we can see that the outer loop iterates p times and the inner loop iterates n times. So the run time complexity of radix sort is given by O(p*n).

If the number of digits in largest element is equal to n, then the run time becomes $O(n^2)$ and this is the worst performance of radix sort. It performs best if the number of digits in largest element is log n, in this case the run time becomes O(n log n). So the radix sort is most efficient when the number of digits in the elements is small. The disadvantage of radix sort is extra space requirement for maintaining queues and so it is not an in-place sort. It is a stable sort.

8.17 Address Calculation Sort

This technique makes use of hashing function(explained in chapter 9) for sorting the elements. Any hashing function f(x) that can be used for sorting should have this property-

If $x < y$, then $f(x) \leq f(y)$

These types of functions are called non decreasing functions or order preserving hashing functions. This function is applied to each element, and according to the value of the hashing function each element is placed in a particular set. When two elements are to be placed in the same set, then they are placed in sorted order. Now we will take some numbers and sort them using address calculation sort.

194, 289, 566, 432, 654, 98, 232, 415, 345, 276, 532, 254, 165, 965, 476

Let us take a function f(x) whose value is equal to the first digit of x, this will obviously be a non decreasing function because if first digit of any number A is less than the first digit of any number B, then A will definitely be less than B.

x	194	289	566	432	654	098	232	415	345	276	532	254	165	965	476
f(x)	1	2	5	4	6	0	2	4	3	2	5	2	1	9	4

Now all the elements will be placed in different sets according to the corresponding values of f(x). The value of function f(x) can be 0, 1, 2....., 9 so there will be 10 sets into which these elements are inserted.

0	098
1	165, 194
2	232, 254, 276, 289
3	345
4	415, 432, 476
5	532, 566
6	654
7	
8	
9	965

We can see that the value of f(x) for the elements 289, 232, 276, 254 is same, so they are inserted in the same set but in sorted order i.e. 232, 254, 276, 289. Similarly other elements are also inserted in their particular sets in sorted order. All the elements in a set are less than elements of the next set and elements in a set are in sorted order. So if we concatenate all the sets then we will get the sorted list.

98, 165, 194, 232, 254, 276, 289, 345, 415, 432, 476, 532, 566, 654, 965

Let us sort the same list by taking another non decreasing hash function-
h(x) = (x/k) * 5 (where k the largest of all numbers)

x	194	289	566	432	654	98	232	415	345	276	532	254	165	965	476
h(x)	1	1	2	2	3	0	1	2	1	1	2	1	0	5	2

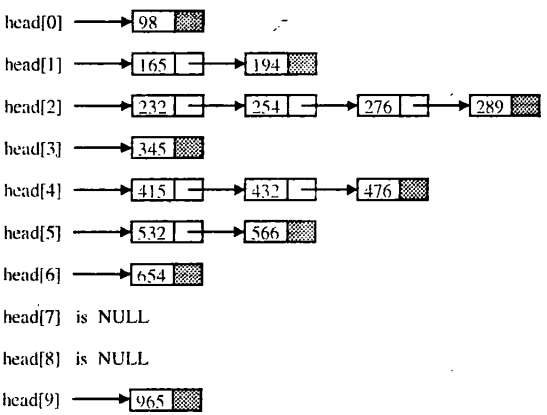
Here the value of function h(x) can be 0, 1, 2, 3, 4, 5 so there will be only 6 sets into which these elements are inserted.

0	98, 165
1	194, 232, 254, 276, 289, 345
2	415, 432, 476, 532, 566
3	654
4	
5	965

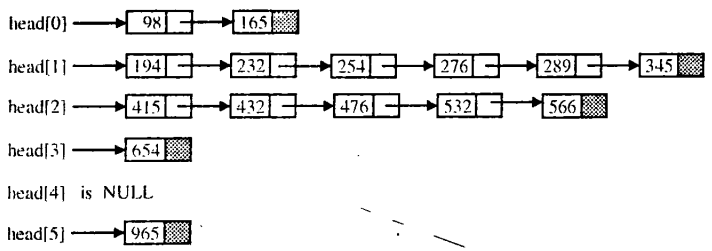
The sorted list that is obtained by concatenating these 6 sets is-
98, 165, 194, 232, 254, 276, 289, 345, 415, 432, 476, 532, 566, 654, 965

For implementing this sorting technique, we can represent each set by a linked list. We know that in each set the elements have to be inserted in sorted order so we will take sorted linked lists. Starting address of each linked list can be maintained in an array of pointers.

In the case of function f(x) which gives us the first digit of an element, there will be 10 linked lists each corresponding to one set. We can take an array of pointers `head[10]`, and each of its element will be a pointer pointing to the first element of these lists. For example `head[0]` will be a pointer that will point to the first element of that linked list in which all elements starting with digit 0 are inserted. Similarly `head[1]`, `head[2]`, ...`head[9]` will be pointers pointing to first elements of other lists.



These linked lists can be easily concatenated to get the final sorted list. In the case of function h(x) discussed before, there will be only 6 linked lists.




```
/*P8.13 Program of sorting using address calculation sort*/
#include<stdio.h>
#include<stdlib.h>
```

```
#define MAX 100
struct node
{
        int info;
        struct node *link;
};
struct node *head[5];
int n,arr[MAX];
int large;
void addr_sort();
void insert(int num,int addr);
int hash_fn(int number);
main()
{
        int i;
        printf("Enter the number of elements in the list : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }/*End of for*/
        for(i=0; i<n; i++)
        {
                if(arr[i] > large)
                        large = arr[i];
        }
        addr_sort();
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d  ",arr[i]);
        printf("\n");
}/*End of main()*/
void addr_sort()
{
        int i,k;
        struct node *p;
        int addr;
        for(i=0; i<=5; i++)
                head[i]=NULL;
        for(i=0; i<n; i++)
        {
                addr = hash_fn(arr[i]);
                insert(arr[i],addr);
        }
        printf("\n");
        for(i=0; i<=5; i++)
        {
                printf("head(%d) -> ",i);
                p = head[i];
                while(p!=NULL)
                {
                        printf("%d ",p->info);
                        p = p->link;
                }
                printf("\n");
        }
        printf("\n");
        /*Taking the elements of linked lists in array*/
        i=0;
        for(k=0; k<=5; k++)
        {
```

```
                p = head[k];
                while(p!=NULL)
                {
                        arr[i++] = p->info;
                        p = p->link;
                }
        }
}/*End of addr_sort()*/
/*Insert the number in sorted linked list*/
void insert(int num,int addr)
{
        struct node *q,*tmp;
        tmp = malloc(sizeof(struct node));
        tmp->info = num;
        /*list empty or item to be added in beginning*/
        if(head[addr] == NULL || num < head[addr]->info)
        {
                tmp->link = head[addr];
                head[addr] = tmp;
                return;
        }
        else
        {
                q = head[addr];
                while(q->link != NULL && q->link->info < num)
                        q = q->link;
                tmp->link = q->link;
                q->link = tmp;
        }
}/*End of insert()*/
int hash_fn(int number)
{
        int addr;
        float tmp;
        tmp = (float)number/large;
        addr = tmp*5;
        return(addr);
}/*End of hash_fn()*/
```

8.17.1 Analysis of Address Calculation Sort

In address calculation sort. we maintain sorted linked lists and so the time requirement is mainly dependent on the insertion time of elements in the lists. If any list becomes too long i.e. most of the elements have to be inserted in the same list then the run time becomes close to $O(n^2)$. If all the elements are uniformly distributed among the lists, i.e. almost each element is inserted in a separate list then only little work has to be done to insert the elements in their respective lists and hence the run time becomes linear i.e. O(n). So the run time does not depend on the original order of the data but it depends on how the hashing function distributes the elements among the lists. If the hashing function is such that many elements are mapped into the same list, then the sort is not efficient. This is not an in-place sort since space is needed for nodes of linked lists and header nodes. It is a stable sort.

```
/*P8.14 Program of sorting records using bubble sort algorithm*/
#include <stdio.h>
#define MAX 100
struct record

        char name[20];
        int age;
        int salary;
;
```

```
main()
{
        struct record arr[MAX],temp;
        int i,j,n, xchanges;
        printf("Enter the number of records : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter record  %d : \n",i+1);
                printf("Enter name : ");
                scanf("%s",arr[i].name);
                printf("Enter age : ");
                scanf("%d", &arr[i].age);
                printf("Enter salary : ");
                scanf("%d", &arr[i].salary);
                printf("\n");
        }
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(arr[j].age > arr[j+1].age)
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp ;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
        printf("List of Records Sorted on age \n");
        for(i=0; i<n; i++)
        {
                printf("%s\t\t",arr[i].name);
                printf("%d\t",arr[i].age);
                printf("%d\n",arr[i].salary);
        }
        printf("\n");
}/*End of main()*/

/*P8.15 Program to sort the records on different keys using bubble sort*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX 100
struct date
{
        int day;
        int month;
        int year;
};
struct employee
{
        char name[20];
        struct date dob;
        struct date doj;
        int salary;
};
void sort_name(struct employee emp[],int n);
void sort_dob(struct employee emp[],int n);
```

```
void sort_doj(struct employee emp[],int n);
void sort_salary(struct employee emp[],int n);
int datecmp(struct date date1,struct date date2);
main()
{
        struct employee emp[MAX];
        int i,n,choice;
        printf("Enter the number of employees : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter name : ");
                scanf("%s",emp[i].name);
                printf("Enter date of birth(dd/mm/yy): ");
                scanf("%d/%d",&emp[i].dob.day,&emp[i].dob.month);
                scanf("/%d",&emp[i].dob.year);
                printf("Enter date of joining(dd/mm/yy) : ");
                scanf("%d/%d",&emp[i].doj.day,&emp[i].doj.month);
                scanf("/%d",&emp[i].doj.year);
                printf("Enter salary : ");
                scanf("%d",&emp[i].salary);
                printf("\n");
        }
        while(1)
        {
                printf("1.Sort by name alphabetically\n");
                printf("2.Sort by date of birth, in descending order\n");
                printf("3.Sort by date of joining, in descending order\n");
                printf("4.Sort by salary in ascending order\n");
                printf("5.Exit\n");
                printf("Enter your choice :");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                sort_name(emp,n);
                                break;
                        case 2:
                                sort_dob(emp,n);
                                break;
                        case 3:
                                sort_doj(emp,n);
                                break;
                        case 4:
                                sort_salary(emp,n);
                                break;
                        case 5:
                                exit(1);
                        default:
                                printf("Wrong choice\n");
                }/*End of switch*/
                for(i=0; i<n; i++)
                {
                        printf("%s\t\t",emp[i].name);
                        printf("%d/%d",emp[i].dob.day,emp[i].dob.month);
                        printf("/%d\t\t",emp[i].dob.year);
                        printf("%d/%d",emp[i].doj.day,emp[i].doj.month);
                        printf("/%d\t\t",emp[i].doj.year);
                        printf("%d\n",emp[i].salary);
                }
                printf("\n");
        }/*End of while*/
}/*End of main()*/
```

```
void sort_name(struct employee emp[],int n)
{
        struct employee temp;
        int i,j,xchanges;
        for(i=0; i<n-1; i++)
        {
                xchanges=0;
                for(j=0; j<n-1-i; j++)
                {
                        if( strcmp(emp[j].name, emp[j+1].name) > 0)
                        {
                                temp = emp[j];
                                emp[j] = emp[j+1];
                                emp[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
}/*End of sort_name()*/
void sort_dob(struct employee emp[],int n)
{
        struct employee temp;
        int i,j,xchanges;
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(datecmp(emp[j].dob,emp[j+1].dob) > 0)
                        {
                                temp = emp[j];
                                emp[j] = emp[j+1];
                                emp[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
}/*End of sort_dob()*/
void sort_doj(struct employee emp[],int n)
{
        struct employee temp;
        int i,j,xchanges;
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(datecmp(emp[j].doj,emp[j+1].doj ) > 0)
                        {
                                temp = emp[j];
                                emp[j] = emp[j+1];
                                emp[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
```

```
}/*End of sort_doj()*/
void sort_salary(struct employee emp[],int n)
{
    struct employee temp;
    int i,j,xchanges;
    for(i=0; i<n-1; i++)
    {
        xchanges = 0;
        for(j=0; j<n-1-i; j++)
        {
            if(emp[j].salary > emp[j+1].salary)
            {
                temp = emp[j];
                emp[j] = emp[j+1];
                emp[j+1] = temp;
                xchanges++;
            }
        }
        if(xchanges == 0)
            break;
    }
}/*End of sort_salary()*/
/*Returns 1 if date1 < date2, returns -1 if date1 > date2, return 0 if equal*/
int datecmp(struct date date1, struct date date2 )
{
    if(date1.year < date2.year)
        return 1;
    if(date1.year > date2.year)
        return -1;
    if(date1.month < date2.month)
        return 1;
    if(date1.month > date2.month)
        return -1;
    if(date1.day < date2.day)
        return 1;
    if(date1.day > date2.day)
        return -1;
    return 0;
}/*End of datecmp()*/

/*P8.16 Program of sorting by address using bubble sort algorithm*/
#include <stdio.h>
#define MAX 100
struct record
{
    char name[20];
    int age;
    int salary;
};
main()
{
    struct record arr[MAX],*ptr[MAX],*temp;
    int i,j,n, xchanges;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
        printf("Enter record  %d : \n",i+1);
        printf("Enter name : ");
        scanf("%s",arr[i].name);
        printf("Enter age : ");
        scanf("%d",&arr[i].age);
```

```
                printf("Enter salary : ");
                scanf("%d",&arr[i].salary);
                printf("\n");
                ptr[i] = &arr[i];
        }
        for(i=0; i<n-1; i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(ptr[j]->age >  ptr[j+1]->age)
                        {
                                temp = ptr[j];
                                ptr[j] = ptr[j+1];
                                ptr[j+1] = temp ;
                                xchanges++;
                        }
                }
                if(xchanges == 0)
                        break;
        }
        printf("List of Records Sorted on age \n");
        for(i=0; i<n; i++)
        {
                printf("%s\t\t", ptr[i]->name);
                printf("%d\t\t",ptr[i]->age);
                printf("%d\n",ptr[i]->salary);
        }
        printf("\n");
}/*End of main()*/
```

Exercise

1. Show with an example that selection sort is not data sensitive.
2. Write recursive program for sorting an array through selection sort.
3. Write a program to sort an array in descending order using selection sort.
4. Modify the selection sort program given in the chapter so that in each pass the larger element moves towards the end.
5. Show the elements of the following array after four passes of the bubble sort.
   34 23 12 9 45 67 21 89 32 10
6. Modify the bubble sort program so that the smallest element is bubbled up in each pass.
7. Modify the bubble sort given in section 8.9 so that in each pass the sorting is done in both directions. i.e. in each pass the largest element is bubbled up and smallest element is bubbled down. This sorting technique is called bidirectional bubble sort or the shaker sort.
8. Show with the help of an example that selection sort is not a stable sort.
9. Show with the help of an example that bubble sort is a stable sort i.e. it preserves the initial order of the elements.
10. The elements of array after three passes of insertion sort are-
    2 5 20 34 13 19 5 21 89 3
    Show the array after four more passes of the insertion sort are finished.
11. Write a function to sort a linked list using insertion sort.
12. Consider the following array of size 10.
    45 3 12 89 54 15 43 78 28 10
    Selection sort, bubble sort and insertion sort were applied to this array and the contents of the array after three passes in each sort are shown below.
    (i) 3 12 15 43 45 28 10 54 78 89

(ii) 3 10 12 89 54 15 43 78 28 45
(iii) 3 12 45 89 54 15 43 78 28 10
Identify the sorting technique used in each case.

13. The insertion sort algorithm in the chapter uses linear search to find the position for insertion. Modify this algorithm and use binary search instead of linear search.
14. Suppose we are sorting an array of size 10 using quicksort. The elements of array after finishing the first partitioning are -
3 2 1 5 8 12 16 11 9 20
Which element(or elements) could be the pivot.
15. Show the contents of the following array after placing the pivot 47 at proper place.
47 21 23 56 12 87 19 35 11 36 72 12
16. Show how this input is sorted using heap sort.
12 45 21 76 83 97 82 54
17. Write a program to sort a list of strings using bubble sort.
18. Show the elements of the following array after first pass of the shell sort with increment 5.
34 21 65 89 11 54 88 23 76 98 17 51 72 91 24 12

Searching and Hashing

9.1 Sequential Search (Linear search)

Sequential search is performed in a linear way i.e. it starts from the beginning of the list and continues till we find the item or reach the end of list. The item to be searched is compared with each element of the list one by one, starting from the first element. For example, consider the array in figure 9.1, suppose we want to search for value 12 in this array. This value is compared with `arr[0]`, `arr[1]`, `arr[2]`,.....`arr[10]`, `arr[11]`. Since the value is found at index 11, the search is successful.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
96	19	85	9	16	29	2	36	41	67	53	12	66	6	75	82	89	23	93	45

Figure 9.1

Now suppose we have to find value 56 in the above array. This value is compared with `arr[0]`, `arr[1]`, `arr[2]`,.....`arr[19]`. The value was not found even after examining all the elements of the array so the search is unsuccessful. The program for sequential search is-

```
/*P9.1 Sequential search in an array*/
#include <stdio.h>
#define MAX 50

main()
{
        int i,n,item,arr[MAX],index;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        printf("Enter the elements : \n");
        for(i=0; i<n; i++)
                scanf("%d", &arr[i]);
        printf("Enter the item to be searched : ");
        scanf("%d", &item);
        index = LinearSearch(arr,n,item);
        if(index == -1)
                printf("%d not found in array\n",item);
        else
                printf("%d found at position %d\n",item,index);
}
int LinearSearch(int arr[],int n,int item)
{
        int i=0;
        while(i<n && item!=arr[i])
                i++;
        if(i<n)
                return i;
        else
                return -1;
}
```

The function `LinearSearch()` returns location of the item if found, or -1 otherwise.

The number of comparisons required to search for an item depends on the position of element inside the array. The best case is when the item is present at the first position and in this case only one comparison is done. The worst case occurs when the item is not present in the array and in this case n comparisons are required where n is the total number of elements. Searching for an item that is present at the i^{th} position requires i comparisons. Now let us find out the average number of comparisons required in a successful search assuming that the probability of searching of all elements is same.

$(1 + 2 + 3 + 4 + \ldots\ldots\ldots + n)/n = (n+1)/2$

So in both average and worst case the run time complexity of linear search is O(n).

This search is good for data structures like linked lists, because no random access to elements is required. Sequential search in linked lists is given in chapter 3.

In the while loop of the `LinearSearch()` we are doing two comparisons in each iteration, the comparison `i<n` can be avoided by using a sentinel value. In the last location of the array i.e. `arr[n]`, we can temporarily place the value of item that is being searched. This value terminates the search when the search item is not present in the array.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Search 12	96	19	85	9	16	29	2	36	41	67	53	12	66	6	75	82	89	23	93	45	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Search 56	96	19	85	9	16	29	2	36	41	67	53	12	66	6	75	82	89	23	93	45	56

The function for sequential search using sentinel is-

```
int LinearSearch(int arr[],int n,int item)
{
        int i=0;
        while(item!=arr[i])
                i++;
        if(i<n)
                return i;
        else
                return -1;
}
```

The number of comparisons in an unsuccessful search can be reduced if the array is sorted. In this case we need not search for the item till the end of the list, we can terminate our search as soon as we find an element that is greater or equal to the search item(in ascending array). If the element is equal to the search item the search is successful otherwise it is unsuccessful. The function for sequential search in an ascending order array is-

```
int LinearSearch(int arr[],int n,int item)
{
        int i=0;
        while(i<n && arr[i]<item)
                i++;
        if(arr[i]==item)
                return i;
        else
                return -1;
}
```

9.2 Binary Search

The prerequisite for binary search is that the array should be sorted. Firstly we compare the item to be searched with the middle element of the array. If the item is found there, our search finishes successfully otherwise the array is divided into two halves, first half contains all elements to the left of the middle element and the other one consists of all the elements to the right side of the middle element. Since the array is sorted, all the elements in the left half will be smaller than the middle element and the elements in the right half will be greater than the

middle element. If the item to be searched is less than the middle element, it is searched in left half otherwise it is searched in the right half.

Now search proceeds in the smaller portion of the array(subarray) and the item is compared with its middle element. If the item is same as the middle element, search finishes otherwise again the subarray is divided into two halves and the search is performed in one of these halves. This process of comparing the item with the middle element and dividing the array continues till we find the required item or get a portion which does not have any element.

To implement this procedure we will take 3 variables viz. `low`, `up` and `mid` that will keep track of the status of lower limit, upper limit and middle value of that portion of the array, in which we will search the element. If the array contains even number of elements, there will be two middle elements, we'll take first one as the middle element. The value of the `mid` can be calculated as-

```
mid = (low+up)/2 ;
```

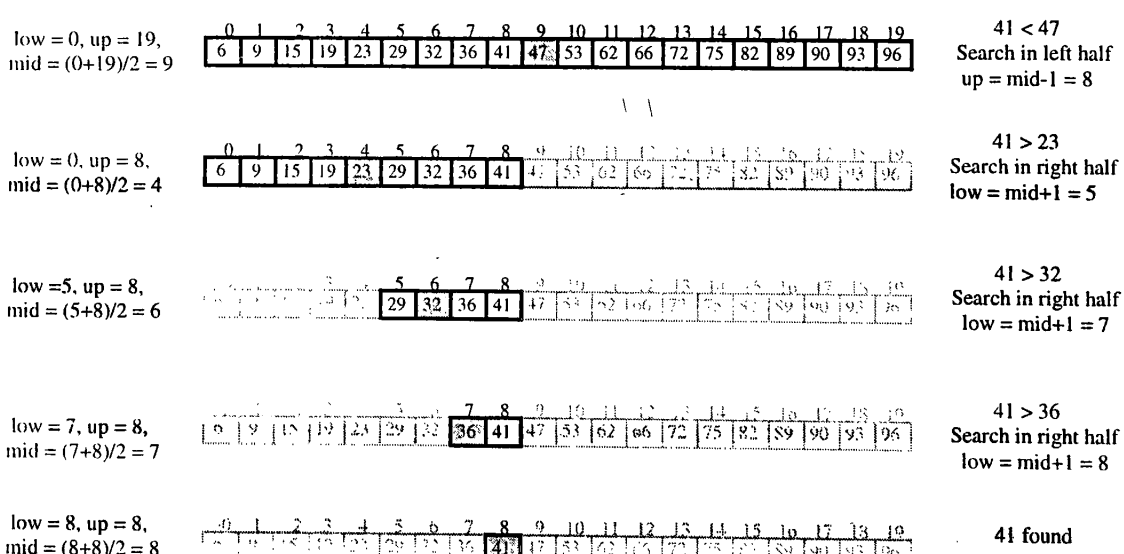
The middle element of the array would be `arr[mid]`, the left half of the array would be `arr[low]`..... `arr[mid-1]` and the right half of the array would be `arr[mid+1]`.... `arr[up]`.

The item is compared with the mid element, if it is not found then the value of `low` or `up` is updated for selecting the left or right half. When `low` becomes greater than `up`, the search is unsuccessful as there is no portion left in which to search.

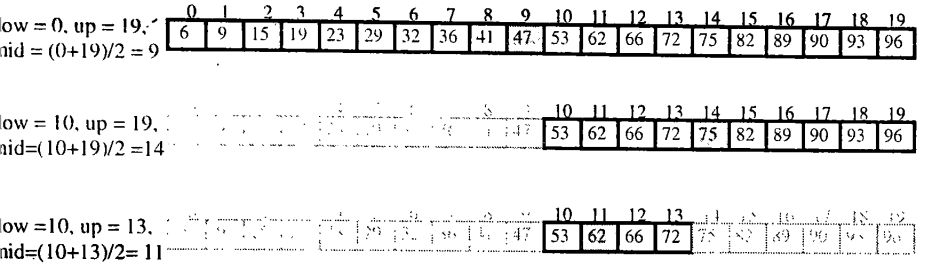
If item > arr[mid]
    Search will resume in right half which is arr[mid+1] .....arr[up]
    So low = mid+1, up remains same
If item < arr[mid]
    Search will resume in left half which is arr[low] .....arr[mid-1]
    up = mid-1, low remains same
If item == arr[mid], search is successful
    Item found at mid position
If low > up , search is unsuccessful
    Item not present in array

Let us take a sorted array of 20 elements and search for elements 41, 62, and 63 in this array one by one. The portion of array in which the element is searched is shown with a bold boundary in the figure.

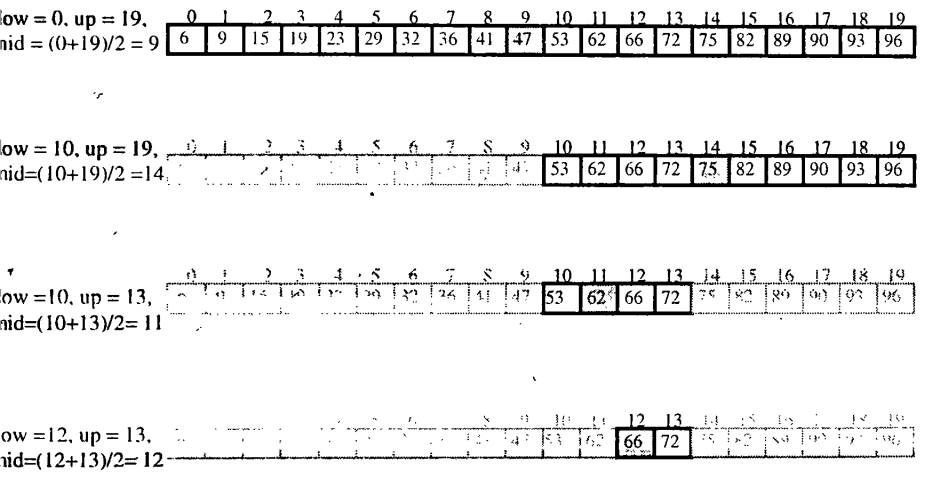
Search 41



arch 62



arch 63



ow low=12 and up=11, the value of low has exceeded the value of up so the search is unsuccessful. The ogram for binary search is-

```
P9.4 Binary search in an array*/
nclude <stdio.h>
efine MAX 50
in()

    int i,size,item,arr[MAX],index;
    printf("Enter the number of elements : ");
    scanf("%d",&size);
    printf("Enter the elements (in sorted order) : \n");

    for(i=0; i<size; i++)
        scanf("%d",&arr[i]);
    printf("Enter the item to be searched : ");
    scanf("%d",&item);
    index = BinarySearch(arr,size,item);
    if(index == -1)
        printf("%d not found in array\n",item);
    else
        printf("%d found at position %d\n",item,index);
```

```
int BinarySearch(int arr[],int size,int item)
{
        int low=0,up=size-1,mid;

        while(low<=up)
        {
                mid = (low+up)/2;
                if(item > arr[mid])
                        low = mid+1;    /*Search in right half */
                else if(item < arr[mid])
                        up = mid-1;     /*Search in left half */
                else
                        return mid;
        }
        return -1;
}
```

The function `BinarySearch()` returns the location of the `item` if found, or -1 otherwise.

The best case of binary search is when the item to be searched is present in the middle of the array, and in this case the loop is executed only once. The worst case is when the item is not present in the array. In each iteration, the array is divided into half, so if the size of array is n, there will be maximum log n such divisions. Thus there will be log n comparisons in the worst case. The run time complexity of binary search is O(log n) and so it is more efficient than the linear search. For an array of about 1000000 elements, the maximum number of comparisons required to find any element would be only 20.

Binary search is preferred only where the data is static i.e. very few insertion and deletions are done. This is because whenever an insertion or deletion is to be done, many elements have to be moved to keep the data in sorted order. Binary search is not suitable for linked list because it requires direct access to the middle element. The recursive function for binary search is-

```
int RBinarySearch(int arr[],int low,int up,int item)
{
        int mid;

        if(low>up)
                return -1;
        mid = (low+up)/2;
        if(item > arr[mid]) /*Search in right half */
                RBinarySearch(arr,mid+1,up,item);
        else if(item < arr[mid])/*Search in left half */
                RBinarySearch(arr,low,mid-1,item);
        else
                return mid;
}
```

9.3 Hashing

We have seen different searching techniques where search time is dependent on the number of elements. Sequential search, binary search and all the search trees are totally dependent on number of elements and many key comparisons are involved. Now we'll see another approach where less key comparisons are involved and searching can be performed in constant time i.e. search time is independent of the number of elements.

Suppose we have keys which are in the range 0 to n-1, and all of them are unique. We can take an array of size n, and store the records in that array based on the condition that key and array index are same. For example, suppose we have to store the records of 15 students and each record consists of roll number and name of the student. The roll numbers are in the range 0 to 14 and they are taken as keys. We can take an array of size 15 and store these records in it as-

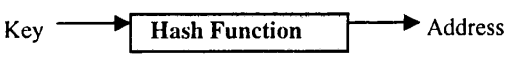
[0]	0	Devanshi
[1]	1	Saachi
[2]	2	Raghav
[3]	3	Supreet
[4]	4	Anushka
[5]	5	Prajwal
[6]	6	Shivani
[7]	7	Parul
[8]	8	Shriya
[9]	9	Samarth
[10]	10	Arnav
[11]	11	Dyuthi
[12]	12	Anjali
[13]	13	Sanjana
[14]	14	Niharika

Figure 9.2

The record with key(roll number) 0 is stored at array index 0, record with key 1 is stored at array index 1 and) on. Now whenever we have to search any record with key k, we can directly go to index k of the array, ecause random access is possible in array. Thus we can access any record in constant time and no key omparisons are involved. This method is known as direct addressing or key-indexed search but it is useful only hen the set of possible key values is small.

Consider the case when we have to store records of 500 employees of a company, and their 6 digit employee l is taken as the key. The employee id can be anything from 000000 to 999999, so here the set of possible key alues is much more than the number of keys. If we adopt the direct addressing method, then we will need an ray of size 10^6 and only 500 locations of this array would be used. In practice, the number of possible key alues will be more than the number of keys actually stored, so this direct addressing is rarely used.

Now let us see how we can modify the direct addressing approach so that there is no wastage of space and ill we can use the value of key to find out its address. For this we will need some procedure through which we an convert the key into an integer within a range, and this converted value can be used as index of the array. nstead of taking key equal to the array index, we can compute the array index from the key. This process of nverting a key to an address(index of array) is called hashing or key to address transformation and it is done rough hash function. A hash function is used to generate an address from a key or we can say that each key is apped on a particular array index through hash function. The hash function takes a key as input and returns the ash value of that key which is used as the address for storing the key in the array. Keys may be of any type like tegers, strings etc but hash value will always be an integer.



e can write this as-

```
(k) = a
```

Here h is the hash function, k is the key and a is the hash value of the key k. Now the key k can be stored at ray index a, which is also known as its home address. This process of generating addresses from keys is lled hashing the key and the array in which insertion and searching is done through hashing is called hash ble. Each entry of hash table consists of a key and the associated record.

For inserting a record, we generate an address(index) by applying hash function to the key, and insert the cord at that address. For accessing any record, we apply the same hash function to the key and then access the cord at the address given by hash function.

Let us take the example of storing records of books where ISBN numbers of the books are keys. Suppose we ke a hash function that maps a key to address by adding the digits of the key. For example if ISBN is 83330487, then the record of this book will be stored at index 45. The figure 9.3 shows the records of 4 books ored in a hash table.

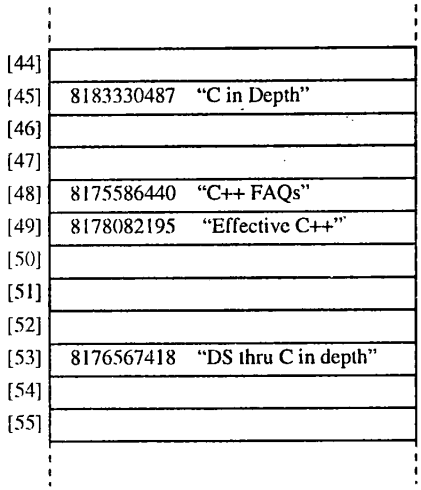


Figure 9.3

Now suppose we have to store a book with ISBN 8173380843. The sum of the digits is 45, i.e. the address given by our hash function is 45, but this address is already occupied. This situation is called **collision.** Collision occurs when the hash function generates the same address for different keys. The keys which are mapped to same address are called **synonyms**. For resolving collision we have different collision resolution techniques about which we will study in detail later in this chapter. Ideally a hash function should give unique addresses for all keys but this is practically not possible. So we should try to select a hash function that minimizes collision. While making algorithms that use hashing we have to mainly focus on these two things-

1. Selecting a hash function that converts keys to addresses.
2. Resolving the collision.

9.3.1 Hash functions

Now we know that hash function generates a table address from a given key. It works like mapping interface between key and hash table. If the size of hash table is m, then we need a hash function that can generate addresses in the range 0 to m-1. Basically we have two criteria for choosing a good hash function-

1. It should be easy to compute.
2. It should generate addresses with minimum collision i.e. it should distribute the keys as uniformly as possible in the hash table.

If we know beforehand about the types of keys that will arrive, then we can select a good hash function that gives minimum collision for these keys. But the nature of keys is generally not known in advance; hence the general approach is to take a function that can work on the key such that addresses generated are distributed randomly.

Now we will see some techniques for making hash functions. In all hashing functions that we will discuss, we will assume key to be an integer. If the key is not an integer then it can easily be converted into one. For example an alphanumeric key can be converted to an integer by using the ASCII values of the characters.

9.3.1.1 Truncation (or Extraction)

This is the easiest method for computing address from a key. Here we take only a part of the key as address, for example we may take some rightmost digits or leftmost digits. Let us take some 8 digit keys and find addresses for them.

82394561, 87139465, 83567271, 85943228

Suppose table size is 100 and we decide to take 2 rightmost digits for getting the hash table address, so the addresses of above keys will be 61, 65, 71 and 28 respectively.

This method is easy to compute but chances of collision are more because last two digits can be same in many keys.

9.3.1.2 Midsquare Method

In mid square method, the key is squared and some digits or bits from the middle of this square are taken as address. Generally the selection of digits depends on the size of the table. It is important that the same digits should be selected from the squares of all the keys. Suppose our keys are 4 digit integers and the size of table is 1000; we will need 3 digit addresses. We can square the keys and take the 3rd, 4th and 5th digits from each squared number as the hash address for the corresponding key.

Key:	1337	1273	1391	1026
Squares of key:	1787569	1620529	1934881	1052676
Address:	875	205	348	526

If the key is too large to square then we can take a part of the key and perform midsquare method on that part rather than on the whole key. For example if we have 7 digit keys, we can take a part of the key which contains first 4 digits of the key and square that part.

9.3.1.3 Folding Method

In this method, the key is broken into different parts where the length of each part is same as that of the required address, except possibly the last part. After this these parts are shifted such that the least significant digits of all parts are in same line and then these parts are added. The address of the key can then be obtained by ignoring the final carry in the sum. Suppose we have a table of size 1000, and we have to find address for a 12 digit key. Since the address should be of 3 digits, we will break the key in parts containing 3 digits.

738239456527 => 738 239 456 527

After this, these parts are shifted and added.

```
  738
  239
  456
  527
 1960
```

Now ignoring the final carry 1, the hash address for the key is 960. This technique is called shift folding; it can be modified to get another folding technique called boundary folding. In this method, the key is assumed to be written on a paper which is folded at the boundaries of the parts of the key, so all even parts are reversed before addition.

```
  738
  932  (Reversed)
  456
  725  (Reversed)
 2851
```

Now ignoring the final carry 2, the hash address for the key is 851.

9.3.1.4 Division Method(Modulo-Divison)

In this method, the key is divided by the table size and the remainder is taken as the address for hash table. In C language, this operation is provided by % operator. This method ensures that we will get the address in the limited range of the table. If the size of table is m, then we will get the addresses in the range {0, 1,....., m-1}.

H(k) = k mod m

Collisions can be minimized if the table size is taken to be a prime number. Let us take table of size 97 and see how the following keys will be inserted in it.

82394561, 87139465, 83567271

82394561 % 97 = 45
87139465 % 97 = 0
83567271 % 97 = 25

So here the hash address of above keys will be 45, 0 and 25.

We can combine other hashing methods with division method and it will ensure that addresses are in the range of hash table.

9.3.2 Collision Resolution

An ideal hash function should perform one to one mapping between set of all possible keys and all hash table addresses but this is almost impossible, and no hash function can totally prevent collisions. A collision occurs whenever a key is mapped to an address that is already occupied, and the different collision resolution techniques suggest for an alternate place where this key can be placed. The two collision resolution techniques that we will study are-

1. Open Addressing (Closed Hashing)
2. Separate Chaining (Open Hashing)

Any of these collision resolution techniques can be used with any hash function.

In the process of searching, the given key is compared with many keys and each key comparison is known as a probe. The efficiency of a collision resolution technique is defined in terms of the number of probes required to find a record with a given key.

9.3.2.1 Open Addressing (Closed Hashing)

In open addressing, the key which caused the collision is placed inside the hash table itself but at a location other than its hash address. Initially a key value is mapped to a particular address in the hash table. If that address is already occupied then we will try to insert the key at some other empty location inside the table. The array is assumed to be closed and hence this method is named as closed hashing. We will study three methods to search for an empty location inside the table.

(i) Linear Probing
(ii) Quadratic Probing
(iii) Double Hashing

9.3.2.1.1 Linear Probing

If address given by hash function is already occupied, then the key will be inserted in the next empty position in the hash table. If the address given by hash function is a and it is not empty, then we will try to insert the key at next location i.e. at address `a+1`. If address `a+1` is also occupied then we will try to insert at next address i.e. `a+2` and we will keep on trying successive locations till we find an empty location where the key can be inserted. While probing the array for empty positions, we assume that the array is closed or circular i.e. if array size is N then after
$(N-1)^{th}$ position, search will resume from 0^{th} position of the array.

If a function h returns address 5 for a key and table size is 11, then we will search for empty locations in this sequence - 5, 6, 7, 8, 9, 10, 0, 1, 2, 3, 4. If location 5 is empty, there will be no collision and we can insert the key there, otherwise we will linearly search these locations and insert the key when we find an empty one. Note that we have taken our array to be closed so after the last location(10^{th}) we probe the first location of array(0^{th}).

Let us take a table of size 11, and insert some keys in it taking a hash function-

h(key) = key % 11

h(29) = 29%11 = 7
h(46) = 46%11 = 2
h(18) = 18%11 = 7
h(36) = 36%11 = 3
h(43) = 43%11 = 10
h(21) = 21%11 = 10
h(24) = 24%11 = 2
h(54) = 54%11 = 10

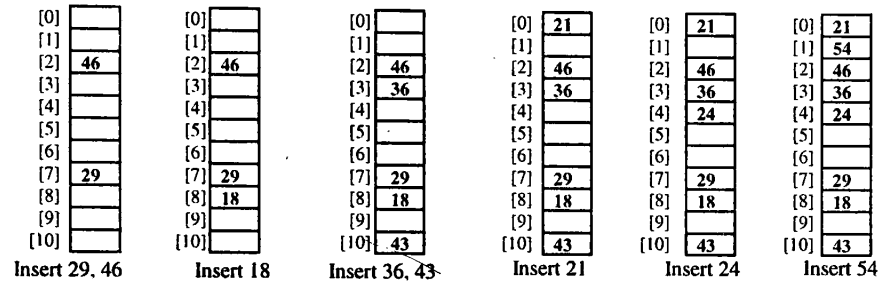


Figure 9.4 Linear Probing

First 29 and 46 are inserted at 7th and 2nd positions of the array respectively. Next 18 is to be inserted but its hash address 7 is already occupied so it will be inserted in the next empty location which is 8th position. After this 36 and 43 are inserted without any collision. Now 21 is to be inserted but its hash address 10 is not empty, so it is inserted in next empty location which is 0th position. Now 24 is to be inserted whose hash address is 2 which is not empty, so next location 3 is probed which is also not empty, and finally it is inserted in the next empty position which is 4th position. In a similar manner 54 is inserted in 1st position.

The formula for linear probing can be written as-

H(k, i) = (h(k) + i) mod Tsize

Here the value of `i` varies from 0 to `Tsize-1`. We have done mod `Tsize` so that the resulting address does not cross the valid range of array indices. So we will search for empty locations in the sequence-

h(key), h(key)+1, h(key)+2, h(key)+3 ..... all modulo Tsize

Let us take the example of inserting 54 in the above table-

H(54, 0) = (10 + 0) % 11 = 10 (not empty)
H(54, 1) = (10 + 1)% 11 = 0 (not empty)
H(54, 2) = (10 + 2)% 11 = 1 (Empty, Insert the key)

This is the way insertion is done in case of linear probing. For searching a key, first we check the hash address position in the table, if key is not available at that position, then we sequentially search the keys after that hash address position. The search terminates if we get the key or reach an empty location or reach the position where we had started. In the last two cases the search is unsuccessful.

The main disadvantage of the linear probing technique is primary clustering. When about half of the table is full, there is tendency of cluster formation i.e. groups of records stored next to each other are created. In the above example a cluster of indices 10, 0, 1, 2, 3, 4 is formed. If a key is mapped to any of these indices then it will be stored at index 5 and the cluster will become bigger. Suppose a key is mapped to index 10, then it will be stored at 5, far away from its home address. The number of probes for inserting or searching this key will be . Clustering increases the number of probes to search or insert a key, and hence the search and insertion times of the records also increase.

9.3.2.1.2 Quadratic Probing

In linear probing, the colliding keys are stored near the initial collision point, resulting in formation of clusters. In quadratic probing this problem is solved by storing the colliding keys away from the initial collision point. The formula for quadratic probing can be written as-

H(k, i) = (h(k) + i^2) mod Tsize

The value of `i` varies from 0 to `Tsize-1` and `h` is the hash function. Here also the array is assumed to be closed. The search for empty locations will be in the sequence-

h(k), h(k)+1, h(k)+4, h(k)+9, h(k)+16, ... all mod Tsize

Let us take a table of size 11 and hash function h(key) = key%11, and apply this technique for inserting th following keys.

h(46) = 46%11 = 2
h(28) = 28%11 = 6
h(21) = 21%11 = 10
h(35) = 35%11 = 2
h(57) = 57%11 = 2
h(39) = 39%11 = 6
h(19) = 19%11 = 8
h(50) = 50%11 = 6

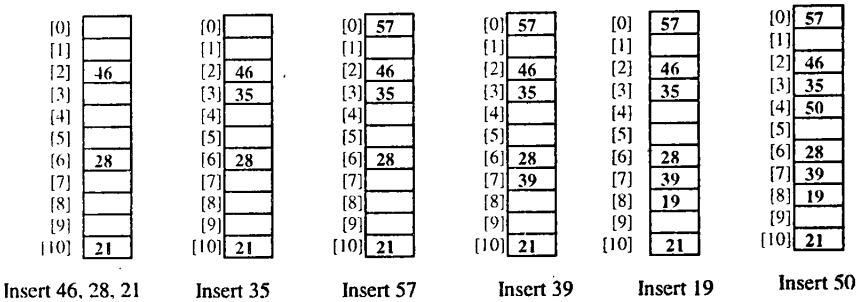


Figure 9.5 Quadratic Probing

Keys 46, 28, 21 are inserted without any collision. Now 35 is to be inserted whose hash address is 2, whic is not empty so next location (2+1)%11 = 3 is tried and it is empty so 35 is inserted in location 3. To insert 57 first location 2 is tried it is not empty so next location (2+1)%11 = 3 is tried, it is also not empty so next locatio (2+4)%11 = 6 is tried, it is also not empty so next location (2+9)%11 = 0 is tried and it is empty so 57 i inserted there. Similarly key 39 is inserted at position 7. The key 19 is inserted without any collision. The ke 50 is inserted at location 4.

H(50, 0) = (6 + 0) % 11 = 6 (not empty)
H(50, 1) = (6 + 1^2) % 11 = 7 (not empty)
H(50, 2) = (6 + 2^2) % 11 = 10 (not empty)
H(50, 3) = (6 + 3^2) % 11 = 4 (Empty, Insert the key)

Quadratic probing does not have the problem of primary clustering as in linear probing, but it gives anothe type of clustering problem known as secondary clustering. Keys that have same hash address will probe th same sequence of locations leading to secondary clustering. For example in the above table, keys 46, 35 and 5 are all mapped to location 2 by the hash function h, and so the locations that will be probed in each case ar same.

46 - 2, 3, 6, 0, 7, 5,.....
35 - 2, 3, 6, 0, 7, 5,.....
57 - 2, 3, 6, 0, 7, 5,.....

In secondary clustering, clusters are not formed by records which are next to each other but they are forme by records which follow the same collision path.

Another limitation of quadratic probing is that it can't access all the positions of the table. An inse operation may fail in spite of empty locations inside the hash table. This problem can be alleviated by takin size of hash table to be a prime number, and in that case at least half of the locations of the hash table will b accessed. In linear probing, an insert operation will fail only when the table is fully occupied.

9.3.2.1.3 Double Hashing

In double hashing, the increment factor is not constant as in linear or quadratic probing, but it depends on th key. The increment factor is another hash function and hence the name double hashing. The formula for doubl hashing can be written as-

H(k, i) = (h(k) + i h'(k)) mod Tsize

The value of `i` varies from 0 to `Tsize-1` and `h` is the hash function, `h'` is the secondary hash function. Th search for empty locations will be in the sequence-

h(k), h(k)+ h'(k), h(k)+ 2 h'(k), h(k)+ 3 h'(k),. ..... all mod Tsize

Let us see how some keys can be inserted in the table taking these two functions-
h(k) = key%11
h'(k) = 7 + (key % 7)

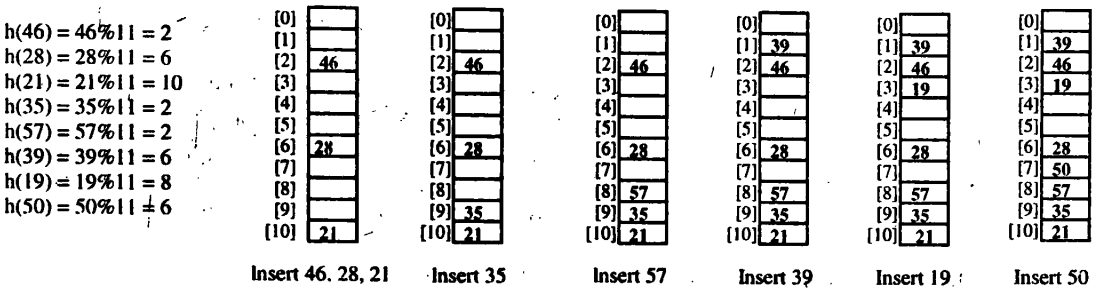


Figure 9.6 Double Hashing

(i) Insertion of 46, 28, 21 - No collision, all are inserted at their home addresses
(ii) Insertion of 35 – Collision at 2.
Next probe is done at - (2 + 1(7-35%7)) %11 = 9, Location 9 is empty, so 35 is inserted there
(iii) Insertion of 57 – Collision at 2
Next probe is done at - (2 + 1(7 - 57%7)) % 11 = 8, Location 8 is empty, so 57 is inserted there
(iv) Insertion of 39 – Collision at 6
Next probe is done at - (6 + 1(7-39%7))%11 = 9, Location 9 is not empty
Next probe is done at - (6 + 2(7-39%7))%11 = 1, Location 1 is empty, so 39 is inserted there
(v) Insertion of 19 – collision at 8
Next probe is done at - (8 + 1(7-19%7))%11 = 10, Location 10 is not empty
Next probe is done at - (8 + 2(7-19%7))%11 = 1, Location 1 is not empty
Next probe is done at - (8 + 3(7-19%7))%11 = 3, Location 3 is empty, so 19 is inserted there
(vi) Insertion of 50 – Collision at 6
Next probe is done at - (6 + 1(7- 50%7))%11 = 1, Location 1 is not empty
Next probe is done at - (6 + 2(7- 50%7))%11 = 7, Location 7 is empty, so 50 is inserted there

The problem of secondary clustering is removed in double hashing because keys that have same hash address probe different sequence of locations. For example 46, 35, 57 all hash to same address 2, but their probe sequences are different.

46 – 2, 5, 8, 0, 3, 6,.....
35 – 2, 9, 5, 1, 8, 4.....
57 – 2, 8, 3, 9, 4, 10.....

While selecting the secondary hash function we should consider these two points – first that it should never give a value of 0, and second that the value given by secondary hash function should be relatively prime to the table size.

Double hashing is more complex and slower than linear and quadratic probing because it requires two times calculation of hash function.

Load factor of a hash table is generally denoted by λ and is calculated as-
$\lambda = n/m$
Here n is the number of records, and m is the number of positions in the hash table.

In open addressing the load factor is always less than 1, as the number of records cannot exceed the size of the table. To improve efficiency, it is desirable that there should always be some empty locations inside the table i.e. the size of table should be more than the number of actual records to be stored. If a table becomes dense i.e. load factor is close to 1, then there will be more chances of collision hence deteriorating the search efficiency. Leaving some locations empty is wastage of space but improves search time so it is a space vs. time tradeoff.

A disadvantage of open addressing is that key values are far from their home addresses, which increases the number of probes. Another problem is that hash table overflow may occur.

9.3.2.1.4 Deletion in open addressed tables

We have seen how to insert and search records in open addressed hash tables, now let us see how records can be deleted. Consider the table given in figure 9.4 and suppose we want to delete the record with key 21. Suppose we do this by marking location 0 as empty(in whichever way the application denotes empty locations) so that it can be used for inserting any other record. Now consider the case when we have to search for key 54. This key maps to address 10, so it is first searched there, and since it is not present there it will be searched in next location which is 0^{th} location. Search will terminate at 0^{th} location because it is empty. So our search for key 54 failed even though it was present in the table. This happened because our search terminated prematurely due to a location that had become empty due to deletion. The same problem can occur in quadratic and double hashing. To avoid this problem we need to differentiate between positions which are empty and positions which previously contained a record but now are empty(vacated) due to deletion. The searching should not terminate when we reach a location that is empty because a record was deleted from there. In the implementation we have done this by a taking a variable `status` in each record.

9.3.2.1.5 Implementation of open Addressed tables

In our program, we will store information of employees in a hash table, and the employee ID of an employee is taken as the key. We will declare a structure `employee` as -

```
struct employee
{
        int empid;
        char name[20];
        int age;
};
```

We will declare another structure named `Record`.

```
struct Record
{
        struct employee info;
        enum type_of_record status;
};
```

Here variable `status` is of type `enum type_of_record` and it can have one of these values - `EMPTY`, `DELETED`, `OCCUPIED`.

After this we will declare an array of type `Record` and this is our hash table.

```
struct Record table[MAX];
```

First of all we need to initialize the locations of the array to indicate that they are empty. This is done by setting the status field of all the array records to EMPTY.

If `table[i]` contains a record then value of `table[i].status` will be `OCCUPIED`, if `table[i]` is empty because of deletion of a record, i.e. it is empty now but was previously occupied then value of `table[i].status` will be `DELETED`, otherwise if `table[i]` is empty(was never occupied) then value of `table[i].status` will be `EMPTY`.

The function `search()` returns -1 if key is not found otherwise it returns the index of the array where key was found. Whenever we search for a key, our search will terminate only when we reach a location `table[i]` whose `status` is `EMPTY`. While inserting, we can stop and insert a new record whenever we reach a location `table[i]` whose `status` is `EMPTY` or `DELETED`. Deletion of a record stored at location `table[i]` is done by setting status to `DELETED`.

The search will terminate when an `EMPTY` location is encountered or the key is found. Thus it is important that at least one location in the table is left `EMPTY` so that the search can terminate if key is not present.

If frequent deletions are performed, then there will be many locations which will be marked as DELETED and this will increase the search time. In this case we'll have to reorganize the table by reinserting all the records in an empty hash table. So if many deletions are to be performed then open addressing is not a good choice. We have used linear probing in the program; it can be easily modified for quadratic or double hashing.

```
/*P9.6 Linear probing*/
#include <stdio.h>
#define MAX 50
enum type_of_record {EMPTY, DELETED,OCCUPIED};
struct employee
{
        int empid;
        char name[20];
        int age;
};
struct Record
{
        struct employee info;
        enum type_of_record status;
};
void insert(struct employee emprec,struct Record table[]);
int search(int key,struct Record table[]);
void del(int key,struct Record table[]);
void display(struct Record table[]);
int hash(int key);
main()
{
        int i,key,choice;
        struct Record table[MAX];
        struct employee emprec;
        for(i=0; i<=MAX-1; i++)
                table[i].status = EMPTY;

        while(1)
        {
                printf("1.Insert a record\n");
                printf("2.Search a record\n");
                printf("3.Delete a record\n");
                printf("4.Display table\n");
                printf("5.Exit\n");
                printf("Enter your choice\n");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1 :
                           printf("Enter empid,name,age : ");
                           scanf("%d%s%d",&emprec.empid,emprec.name,&emprec.age);
                           insert(emprec,table);
                           break;
                        case 2 :
                           printf("Enter a key to be searched : ");
                           scanf("%d",&key);
                           i = search(key,table);
                           if(i==-1)
                                printf("Key not found\n");
                           else
                                printf("Key found at index %d\n",i);
                           break;
                        case 3:
                           printf("Enter a key to be deleted\n");
                           scanf("%d",&key);
                           del(key,table);
```

```
                    break;
                case 4:
                    display(table);
                    break;
                case 5:
                    exit(1);
            }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
int search(int key,struct Record table[])
{
        int i,h,location;
        h = hash(key);
        location = h;

        for(i=1; i!=MAX-1; i++)
        {
                if(table[location].status == EMPTY)
                        return -1;
                if(table[location].info.empid == key)
                        return location;
                location = (h+i)%MAX;
        }
        return -1;
}/*End of search()*/
void insert(struct employee emprec, struct Record table[])
{
        int i,location,h;
        int key = emprec.empid;        /*Extract key from the record*/
        h = hash(key);
        location = h;

        for(i=1; i!=MAX-1; i++)
        {
         if(table[location].status==EMPTY || table[location].status==DELETED)
         {
                table[location].info = emprec;
                table[location].status = OCCUPIED;
                printf("Record inserted\n\n");
                return;
         }
         if(table[location].info.empid == key)
         {
                printf("Duplicate key\n\n");
                return;
         }
         location = (h+i)%MAX;
        }
        printf("Record can't be inserted : Table overFlow\n\n");
}/*End of insert()*/
void display(struct Record table[])
{
        int i;
        for(i=0; i<MAX; i++)
        {
                printf("[%d]  : ",i);
                if(table[i].status==OCCUPIED)
                {
                 printf("Occupied:%d  %s",table[i].info.empid,table[i].info.name);
                 printf("   %d\n", table[i].info.age);
                }
                else if(table[i].status==DELETED)
                        printf("Deleted\n");
```

```
                else
                        printf("Empty\n");
        }
}/*End of display()*/
void del(int key,struct Record table[])
{
        int location = search(key, table);
        if(location == -1)
                printf("Key not found");
        else
                table[location].status = DELETED;
}/*End of del()*/
int hash(int key)
{
        return(key%MAX);
}/*End of hash()*/
```

9.3.2.2 Separate chaining

In this method, linked lists are maintained for elements that have same hash address. Here the hash table does not contain actual keys and records but it is just an array of pointers, where each pointer points to a linked list. All elements having the same hash address i will be stored in a separate linked list, and the starting address of that linked list will be stored in the index i of the hash table. So array index i of the hash table contains a pointer to the list of all elements that share the hash address i. Each element of linked list will contain the whole record with key. These linked lists are referred to as chains and hence the method is named as separate chaining. Let us take an example and see how collisions can be resolved using separate chaining. Suppose we take a hash table of size 7 and hash function H(key) = key%7.

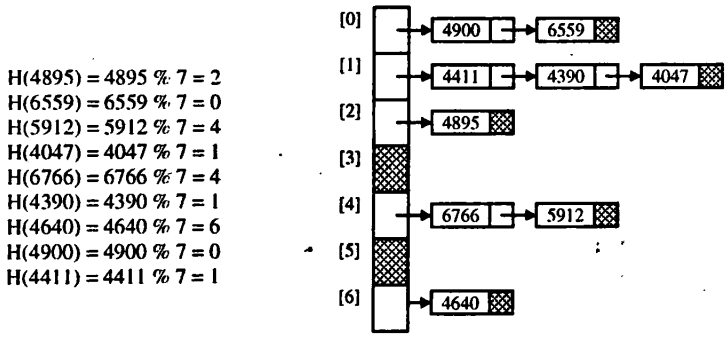


Figure 9.7 Separate Chaining

Here keys 4411, 4390, 4047 all hash to the same address i.e. array index 1, so they are all stored in a separate linked list whose starting address is stored in location 1 of the array. Similarly other keys are also stored in respective lists depending on their hash addresses.

If linked lists are short, performance is good but if lists become long then it takes time to search a given key in any list. To improve the retrieval performance we can maintain the lists in sorted order.

For inserting a key, first we will get the hash value through hash function, then that key will be inserted in the beginning of the corresponding linked list. Searching a key is also same, first we will get the hash value through hash function, and then we will search the key in corresponding linked list. For deleting a key, first that key will be searched and then the node holding that key will be deleted from its linked list.

In open addressing, accessing any record involved comparisons with keys which had different hash values which increased the number of probes. In chaining, comparisons are done only with keys that have same hash values.

In open addressing, all records are stored inside the hash table itself so there can be problem of hash table overflow and to avoid this, enough space has to be allocated at the compilation time. In separate chaining, there will be no problem of hash table overflow because linked lists are dynamically allocated so there is no limitation on the number of records that can be inserted. It is not necessary that the size of table be more than the number of records. Separate chaining is best suited for applications where the number of records is not known in advance.

In open addressing, it is best if some locations are always empty. If records are large then this results in wastage of space. In chaining there is no wastage of space because the space for records is allocated when they arrive.

Implementation of insertion and deletion is simple in separate chaining. The main disadvantage of separate chaining is that it needs extra space for pointers. If there are n records and the table size is m, then we need extra space for n+m pointers. If the records are very small then this extra space can prove to be expensive.

In separate chaining the load factor denotes the average number of elements in each list and it can be greater than 1.

```
/*P9.9 Separate chaining*/
#include <stdio.h>
#include<stdlib.h>
#define MAX 11

struct employee
{
        int empid;
        char name[20];
        int age;
};
struct Record
{
        struct employee info;
        struct Record *link;
};
void insert(struct employee emprec, struct Record *table[]);
int search(int key, struct Record *table[]);
void del(int key, struct Record *table[]);
void display(struct Record *table[]);
int hash(int key);
main()
{
        struct Record *table[MAX];
        struct employee emprec;
        int i,key,choice;
        for(i=0;i<=MAX-1;i++)
                table[i] = NULL;

        while(1)
        {
                printf("1.Insert a record\n");
                printf("2.Search a record\n");
                printf("3.Delete a record\n");
                printf("4.Display table\n");
                printf("5.Exit\n");
                printf("Enter your choice\n");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1 :
                             printf("Enter the record\n");
                             printf("Enter empid, name, age : ");
                             scanf("%d%s%d",&emprec.empid,emprec.name,&emprec.age);
                             insert(emprec,table);
```

```
                    break;
               case 2 :
                    printf("Enter a key to be searched : ");
                    scanf("%d",&key);
                    i = search(key,table);
                    if(i==-1)
                      printf("Key not found\n");
                    else
                      printf("Key found in chain %d\n",i);
                    break;
               case 3:
                    printf("Enter a key to be deleted\n");
                    scanf("%d",&key);
                    del(key,table);
                    break;
               case 4:
                    display(table);
                    break;
               case 5:
                    exit(1);
          }
     }
}/*End of main()*/
void insert(struct employee emprec,struct Record *table[])
{
     int h,key;
     struct Record *tmp;

     key = emprec.empid;  /*Extract the key from the record*/
     if(search(key, table)!=-1)
     {
          printf("Duplicate key\n");
          return;
     }
     h = hash(key);

     /*Insert in the beginning of list h*/
     tmp=malloc(sizeof(struct Record));
     tmp->info = emprec;
     tmp->link = table[h];
     table[h] = tmp;
}/*End of insert()*/
void display(struct Record *table[])
{
     int i;
     struct Record *ptr;

     for(i=0; i<MAX; i++)
     {
       printf("\n[%d]  ", i);
       if(table[i]!=NULL)
       {
            ptr = table[i];
            while(ptr!=NULL)
            {
                 printf("%d %s %d\t",ptr->info.empid,ptr->info.name,ptr->info.age);
                 ptr = ptr->link;
            }
       }
     }
     printf("\n");
}/*End of display()*/
int search(int key,struct Record *table[])
```

```
{
        int h;
        struct Record *ptr;
        h = hash(key);
        ptr = table[h];
        while(ptr!=NULL)
        {
                if(ptr->info.empid == key)
                        return h;
                ptr = ptr->link;
        }
        return -1;
}/*End of search()*/
void del(int key,struct Record *table[])
{
        int h;
        struct Record *tmp, *ptr;

        h = hash(key);
        if(table[h]==NULL)
        {
                printf("Key %d not found\n",key);
                return;
        }
        if(table[h]->info.empid == key)
        {
                tmp=table[h];
                table[h]=table[h]->link;
                free(tmp);
                return;
        }
        ptr = table[h];

        while(ptr->link!=NULL)
        {
                if(ptr->link->info.empid == key)
                {
                        tmp=ptr->link;
                        ptr->link=tmp->link;
                        free(tmp);
                        return;
                }
                ptr=ptr->link;
        }
        printf("Key %d not found\n",key);
}/*End of del()*/
int hash(int key)
{
        return(key%MAX);
}/*End of hash()*/
```

9.3.3 Bucket Hashing

This technique postpones collisions but does not resolve them completely. Here hash table is made up of buckets, where each bucket can hold multiple records. There is one bucket at each hash address and this means that we can store multiple records at a given hash address. Collisions will occur only after a bucket is full. In the example given below we have taken a hash table in which each bucket can hold three records. So we can store three records at the same hash address without collision. A collision occurs only when a fourth record with the same hash address arrives.

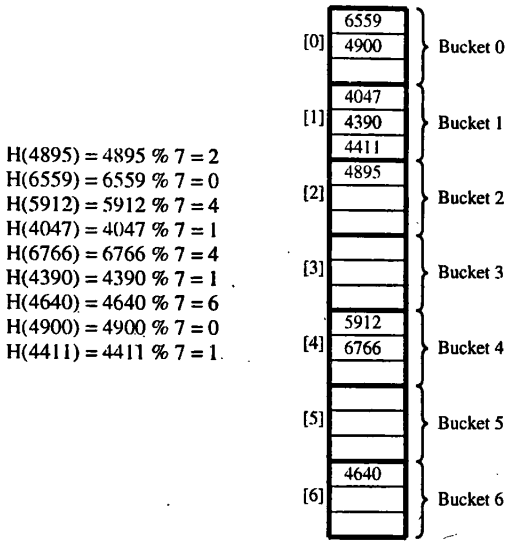


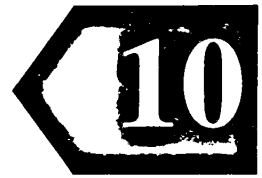
Figure 9.8 Bucket Hashing

A disadvantage of this technique is wastage of space since many buckets will not be occupied or will be partially occupied. Another problem is that this method does not prevent collisions but only defers them, and when collisions occur they have to be resolved using a collision resolution technique like open addressing.

Exercise

1. An array contains the following 18 elements.
   12 19 23 27 30 34 45 56 59 61 76 79 83 85 88 90 94 97
   Perform binary search to find the elements 27, 32, 61, 97 in the array. Show the values of up, mid and low in each step.
2. Find the hash addresses of the following keys using the midsquare method, if the size of table is 1000.
   342, 213, 432, 542, 132, 763, 298
3. Find the hash addresses of the following keys using the folding method, if the size of table is 1000.
   321982432, 213432183, 343541652, 542313753
4. Find the hash addresses of the keys in exercise 3 using the boundary folding method, if the size of table is 1000.
5. How many collisions occur if the hash addresses are generated using the modulo division method, where the table size is 64.
   9893, 2341, 4312, 7893, 4531, 8731, 3184, 5421, 4955,1496
   How many collisions occur if the table size is changed to 67.
6. Insert the following keys in an array of size 17 using the modulo division method. Use linear probing to resolve collisions. 94, 37, 29, 40, 84, 88, 102, 63, 67, 120, 122
7. Insert the following keys in an array of size 17 using the modulo division method. Use quadratic probing to resolve collisions.
   94, 37, 29, 40, 84, 88, 102, 63, 67, 120, 122
8. Insert the following keys in an array of size 17 using the modulo division method. Use double hashing to resolve collisions. Take h'(k) = (key%7) + 1 as the second hash function.
   94, 37, 29, 40, 84, 88, 102, 63, 67, 120, 122
9. What is the length of the longest chain if the following keys are inserted in a table of size 11 using modulo division and separate chaining.
   1457 2134 8255 4720 6779 2709 1061 3213

Storage Management



We have seen data structures which are used to implement different concepts in a programming language. In each one of them we specify the need of storage space for implementing these data structures. For example in the case of linked list, we allocate the space for node at the time of insertion and free the storage space at the time of deletion. This allocation and freeing of space can be done at run time because C language supports dynamic memory allocation. Generally it is the work of operating system to provide the specified memory to user and manage the allocation and release process. Now we will see different techniques and data structures for storage management.

The dynamic memory allocation requires allocation and release of different size of memory at different times. Many applications might be running on a system and they can request for different size of memory. The memory management system of the operating system is responsible for managing the memory and fulfilling the memory requirements of the users.

Suppose we have 512K of available memory and the programs P1, P2, P3, P4 request for memory blocks of sizes 110K, 90K, 120K, 110K respectively. The memory is allocated sequentially to all these programs.

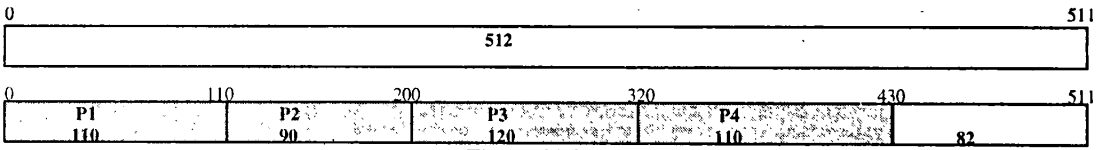


Figure 10.1

After some time programs P1 and P3 release their memory.

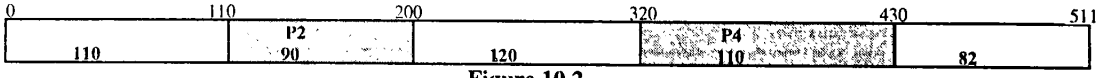


Figure 10.2

Now there are two blocks of memory that are being used, and three blocks that are free; the free blocks can be used to satisfy memory requests. The operating system needs to keep record of all the free memory blocks so that they can be allocated whenever required. This is done by maintaining a linked list of all the free blocks, known as **free list**. Each free block contains a link field that contains the address of next free block. The blocks can be of varying sizes, so a size field is also present in each free block. These fields can be present in some fixed location of the block, so that they can be accessed if the starting address of the block is known. Generally these fields are stored at the start of the block. A pointer named `freeblock` or `avail` is used to point to the first free block of this linked list. The figure 10.3 shows how the free blocks are linked together to form a free list.

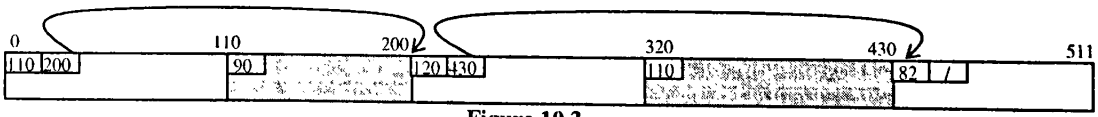


Figure 10.3

Whenever a request for a block of memory comes, memory is allocated from any of the blocks in free list depending upon the size of the memory requested and the sequential fit method used. If the size of block is equal to the memory requested then the whole block is allocated otherwise a portion of the block is allocated and the remaining portion becomes a free block and it remains in the free list. When a block of memory becomes free, it is joined to this free list.

10.1 Sequential Fit Methods

There are many methods for selecting a free block from the free list during allocation of memory requested by the user. The three methods which we'll see are first fit, best fit and worst fit methods. Consider the situation in the figure 10.4, and suppose that a program P8 requests for 25K memory. Now let us see how this memory will be allocated using different sequential fit methods.

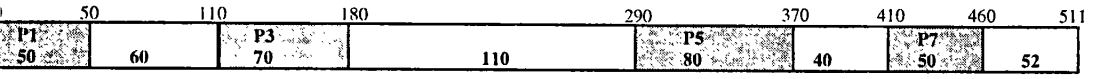


Figure 10.4

10.1.1 First Fit

In this method, the free list is traversed linearly and memory is allocated from the first free block that is large enough to fulfill the request i.e. the first free block whose size is greater or equal to the size of memory requested. If the size of the block is equal to the memory requested, the whole block is allocated otherwise it is split into two blocks, one of which is allocated and the other is put on the free list. In figure 10.4, the first free block that has size greater than the size of memory requested is the block of size 60. So this block is splitted and a portion of this block is allocated for P8 while the remaining block of 35K remains on the free list.

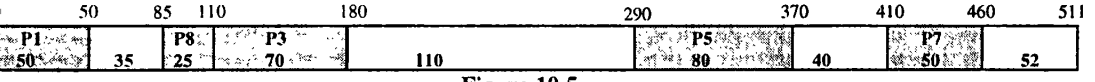


Figure 10.5

Since the link part in each block is stored at the starting of each block, it is better to allocate the second portion of the block for the memory requested. By doing this, there would be no need to change the link part of previous free block. The link part of the splitted free block would become the link part of the new free block and so there is no need to change it.

10.1.2 Best Fit method

In this method, the whole list is traversed and the memory is allocated from the free block whose size is closest to the size of the memory requested i.e. from the smallest free block whose size is greater than or equal to the memory requested. In figure 10.4, the free block of size 40 is the one that is closest to size 25, so a portion from this block is allocated.

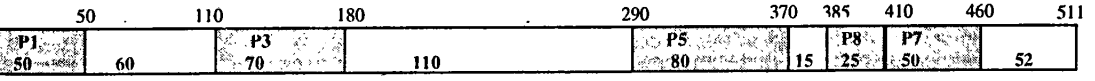


Figure 10.6

The advantage of best fit method is that the bigger blocks are not broken. A disadvantage is that there are many small sized blocks left(here 15K) which are practically unusable.

10.1.3 Worst Fit Method

In this method, the whole list is traversed and memory is always allocated from the free block that is largest in size. In figure 10.4, the largest free block is block of size 110, so a portion from this block is allocated.

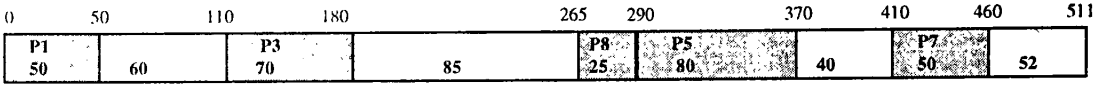


Figure 10.7

The advantage of this method is that after allocation of memory, the blocks that are left are of reasonable size and can be used later. For example here the block left after allocation is of size 85.
The order in which blocks are stored on the free list can improve the searching time. For example the search time in best fit can be reduced if the blocks are arranged in order of increasing size. If the blocks are arranged in order of decreasing size then the largest block will always be first one and so there will be no need of searching in the case of worst fit. For first fit, the blocks can be arranged in order of increasing memory address. Generally the first fit method proves to be the most efficient one and is preferred over others.

10.2 Fragmentation

After several allocations and deallocations, we can end up with memory broken into many small parts which are practically unusable, thus wasting memory. This wastage of memory is known as fragmentation. There are two types of fragmentation problems -

(i) External Fragmentation - This occurs when there are many non contiguous free memory blocks. This results in wastage of memory outside allocated blocks.

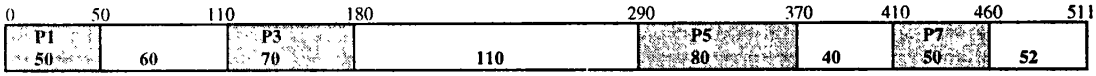


Figure 10.8

In figure 10.8, we have four free memory blocks and the total free memory is 262. Suppose a program requests for memory of size 150. Although the free memory is much more than the memory requested, this requirement cannot be fulfilled because the free memory is divided into non contiguous blocks and we don't have 150K of contiguous free memory. External fragmentation can be removed by compaction.

(ii) Internal Fragmentation - This occurs when memory allocated is more than the memory requested, thus wasting some memory inside the allocated blocks. For example suppose the memory is to be allocated only in sizes of powers of two, i.e. the blocks that can be allocated can be of sizes 1, 2, 4, 8, 16, 32, 64, 128..... Suppose a request of memory of size 100 comes, then the memory block of size 128 would be allocated for that, thereby wasting 28 memory locations inside the allocated block.

10.3 Freeing memory

Now let us see what needs to be done when a memory block is freed. Whenever a memory block is freed, it has to be added to the free list. If the free list is not ordered, then it is added at the front of the list otherwise it is put at the appropriate place in the list. This way after some time there will be lot of small free blocks on the free list and it won't be possible to fulfill a request for a large memory block. At the time of allocation the bigger blocks are splitted into smaller blocks, so while freeing memory there should be some combination procedure that combines small free blocks into large blocks.
If any neighbor (left or right or both) of the freed block is free, then we can remove it from the free list and combine these contiguous free blocks to form a larger free block and put this larger free block on the free list. To find out whether a free neighbor exists, we have to traverse the whole free list. Consider the situation given in figure 10.9.

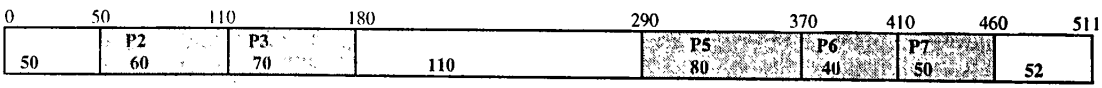


Figure 10.9

Suppose now P5 is freed, we can remove block at address 180 from the free list and combine these two blocks to form one block and put this larger block of size 190 on the free list.

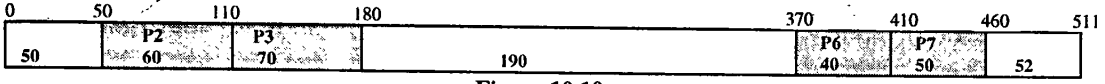


Figure 10.10

If the free list is arranged in order of increasing memory address, then we need not search the whole list. For example suppose we free a block B. We search the list for a block B1 where the address of B1 is greater than address of B.

If the free block B1 is contiguous to B, then B1 is removed from the list and is combined with B. The new combined block is placed in the list at the place of B1. If the block B1 is not contiguous to block B then block B is inserted in the list just before B1.

Suppose block B2 is the free block immediately preceding B1. If B2 is contiguous to block B, then B2 is removed from the list and is combined with B and the new combined block is placed in the list at the place of B2.

10.4 Boundary tag method

In the boundary tag method, there is no need to traverse the free list for finding the address and free status of adjacent free blocks. To achieve this we need to store some extra information in all the blocks.

When a block is freed we need to locate its left and right neighbors, and find whether they are free or not. Let us see how we can calculate the address of left and right neighbor of a block B.

Start address of right neighbor of B = Start address of B + size of B
Start address of left neighbor of B = Start address of B – size of left neighbor

For example in figure 10.11, address of right neighbor of P3 is (110+70) and address of its left neighbor is (110 – 60).

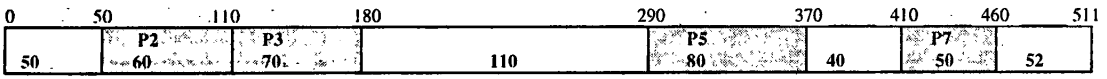


Figure 10.11

Let us assume that a `size` field is stored at a positive offset from the start address so that we can find size by expression `size(p)` where `p` is the start address. Now suppose a block B at address `k` is freed, its size would be `size(k)` and the starting address of right neighbor would be `k+size(k)`. To find start address of left neighbor we need size of left neighbor, but to access its size we need to know its start address. We only know the end address of left neighbor which is `(k-1)` so there is no way in which we can find the size of left neighbor. The solution to this problem is to store a duplicate size field in each block at a negative offset from the end of the block, let us call it `bsize`. If end address is `q` then the size of block can be accessed by expression `bsize(q)`. So now start address of left neighbor of B would be `k-bsize(k-1)`.

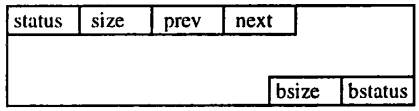
When a block is freed we need to know whether its left and right neighbors are free or not. A status field is stored in each block, if the block is free then this field is 0 otherwise it is 1. This status field is stored both at the start and end of the block. If start address of a block is `p`, then we can access its free status field by `status(p)`, and if `q` is the end address then we can get its free status field by `bstatus(q)`. So if a block B at start address `k` is being freed, we can access status field of right neighbor by `status(k+size(k))` and that of left neighbor by `bstatus(k-1)`.

Each block needs the size and status fields at both the ends i.e. at both the boundaries and hence the name of this method is boundary tag method.

In boundary tag method, the free list will be stored as a double linked list. This is because we don't reach a block by traversal and hence we don't know its predecessor which is required for the removal of a block from

the list. So each free block will have `next` and `prev` pointers pointing to the next and previous free blocks, and these will be stored at a positive offset from the start of the block.

The structure of a block in boundary tag method is-



Whenever a block is freed, first its free status is set to 0, and then statuses of its left and right neighbors are checked. We can combine the freed block with next or previous block or both, if they are free. The free list would be maintained as a circular double linked list with header. Now let us take some examples. Suppose we have 86K memory, and some of it is allocated to programs P1, P2, P3 and P4. The free list is shown in the figure 10.12.

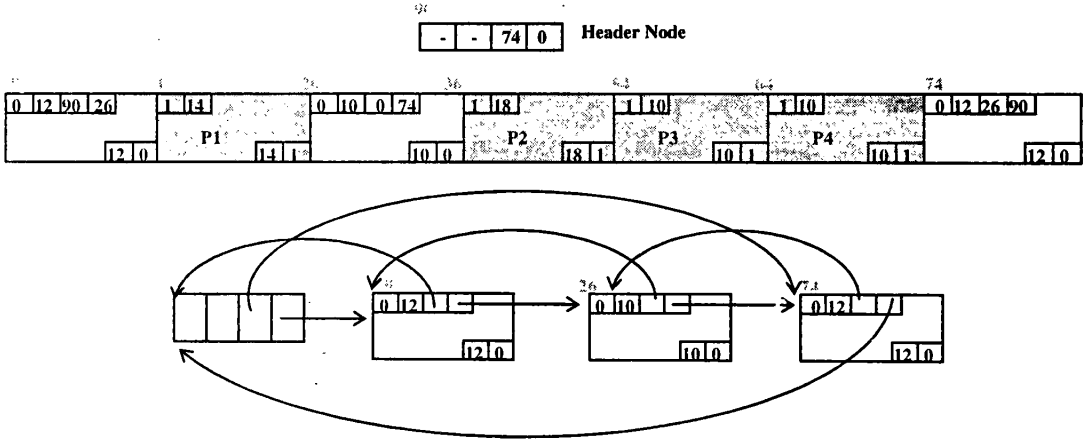


Figure 10.12

Now we'll free memory blocks one by one and see how the free blocks are combined in the free list.

(i) Free P3 in figure 10.12.

P3 has no free neighbor, so it is just added to the beginning of the free list.

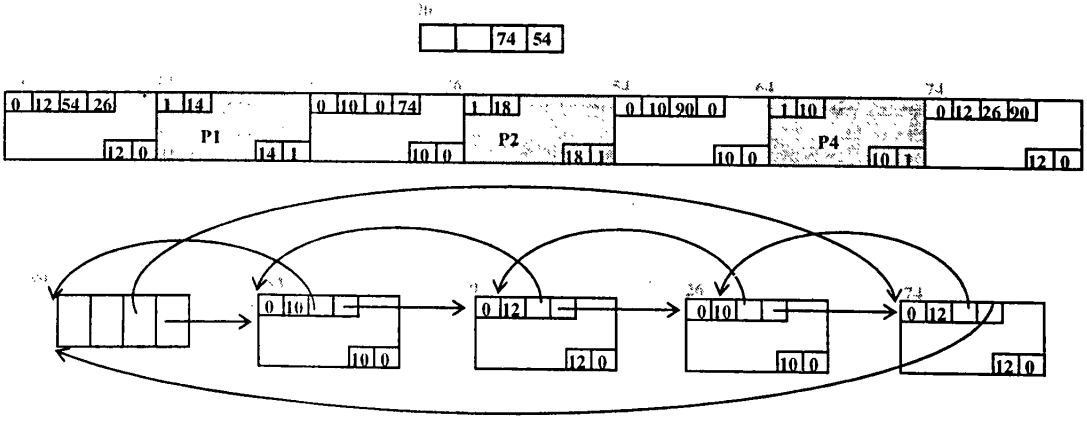


Figure 10.13

i) Free P4 in figure 10.12.

he left neighbor at 54 is not free, but the right neighbor at 74 is free so the block at 74 is removed from the free st and is combined with the freed block and the new block of size 22 is added to the free list.

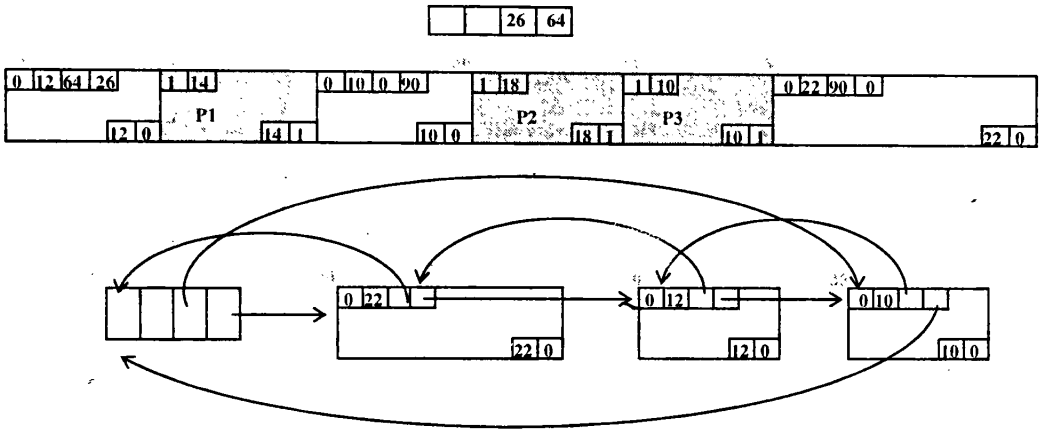


Figure 10.14

ii) Free P2 in figure 10.12

he right neighbor at 54 is not free, but the left neighbor at 26 is free so the block at 26 is removed from the free st and is combined with the freed block and the new block of size 28 is added to the free list.

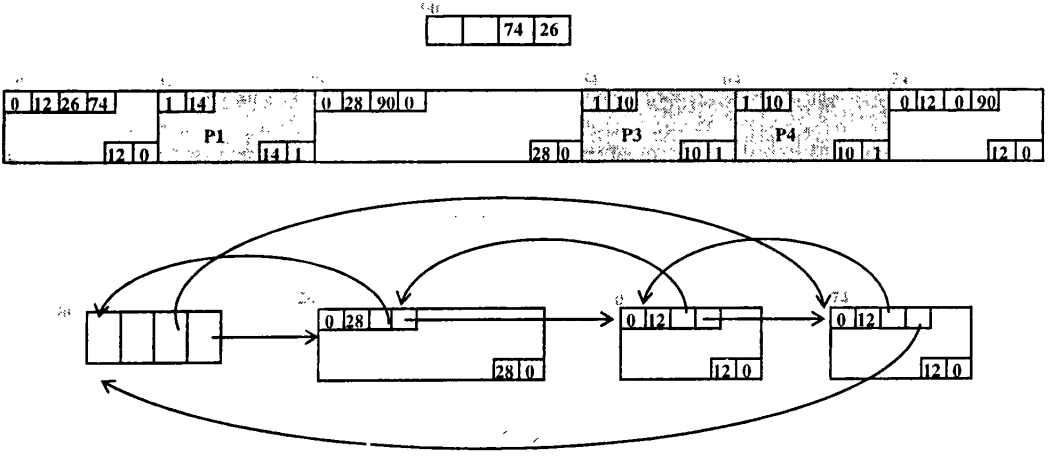


Figure 10.15

v) Free P1 in figure 10.12

he left neighbor is at 0 and right neighbor is at 26, and both are free. So both these blocks are removed from e free list and the new combined block of size 36 is added to the free list.

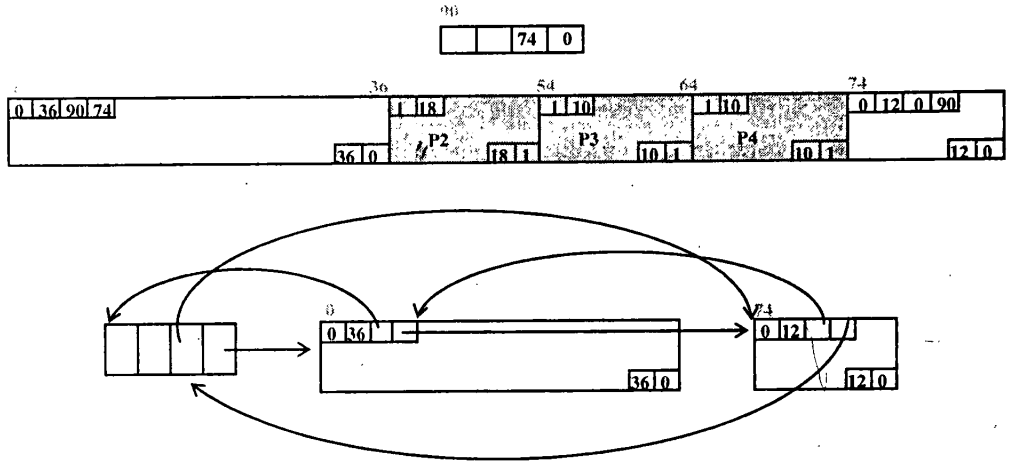


Figure 10.16

10.5 Buddy Systems

In buddy systems, the size of memory blocks that can be allocated is restricted to some fixed set i.e. we can have blocks of only some specific sizes. There is a separate free list for each permitted block size. So in a buddy system, many free lists are maintained where each free list is a double linked list that contains free blocks of same size. Any block can be splitted to form two smaller blocks of permitted sizes, and these blocks are called buddies. Any two buddies i.e. blocks which were splitted from the same parent block can be combined to rebuild the parent block.

When a memory of size n is requested, a block of size p is allocated where p is the smallest permitted block size which is greater or equal to n. If a block of size p is not free then the next larger block is splitted into two blocks called buddies. One of these blocks is added to the appropriate free list and the other block is either allocated(if it is of size p) or again splitted. The process of splitting continues till we get a block of size p.

When a block is freed, first its buddy is checked and if the buddy is also free, these two buddies are combined to form a larger block. Two free blocks can be combined only when they belong to the same parent block i.e. only buddies can be combined. If the buddy of the combined block is also free then it is also combined with its buddy to form a larger block. This combining process continues till there are no buddies left to combine or we get the largest block after combining. The two common buddy systems are binary buddy system and Fibonacci buddy system.

10.5.1 Binary Buddy System

In this system, all block sizes are in powers of 2. So the permitted block sizes in this system are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 and so on. At the time of split, a block of size 2^i is split into two equal blocks of size 2^{i-1} and at the time of combining, two buddies of size 2^{i-1} are combined to form a block of size 2^i.

For all free blocks, double linked lists will be maintained based on power of 2. An i-block is a block of size 2^i, and the free list that contains all free i-blocks is called an i-list. For example a 6-block is a block of size 64 (2^6) and a 6-list is a free list of all free 6-blocks.

Suppose initially the total memory available is 2^m. The permitted size of blocks will be 1, 2, 4, 16,.....2^m. The free lists that will be maintained are: 0-list, 1-list, 2-list.....m-list. Initially the whole memory will be considered as one free block of size 2^m, and this block will be present in the m-list, all other free lists will be empty.

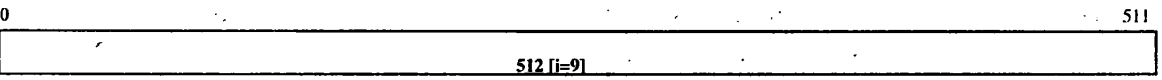
If we want to allocate a memory of size n, then we will find an integer p where $2^{p-1} < n <= 2^p$. Now we know that we need to allocate a p-block. We will look in the p-list, and if it is not empty then we will remove a block from it and allocate it. If the p-list is empty, then we'll look at the (p+1)-list. If (p+1)-list is not empty then we'll

remove a block from it and break it into two buddy blocks where each buddy is a p-block, i.e. each buddy is of size 2^p. Now one of these buddies will be allocated and the other will be put on the p-list. If both p-list and (p+1)-list are empty then we will look at the (p+2)-list and take a block from it and split it into two (p+1) blocks. One of these blocks is put into (p+1)-list and the other is again split into two p-blocks. One of these p-blocks is put into p-list and the other is allocated. This procedure of splitting will continue till we get a block of size p.

When a block of size 2^i is deallocated, we compute the address of its buddy block. If the buddy block is not free then the deallocated block is added to the i-list. If the buddy block is free then we combine the two buddies to form a block of size 2^{i+1}. After combining we find the address of the buddy of the combined block of size 2^{i+1}. If the buddy of this block is not free then this block is added to the (i+1)-list, otherwise it is combined with its buddy to form a block of size 2^{i+2}. This process continues till we get a block whose buddy is not free or till we get a block of size 2^m.

Suppose we have 2^9 = 512 K of memory, and the smallest size of block that can be allocated is taken as 2^3 = 8K. So we will have to maintain 7 free lists viz. 3-list, 4-list, 5-list, 6-list, 7-list, 8-list and 9-list.

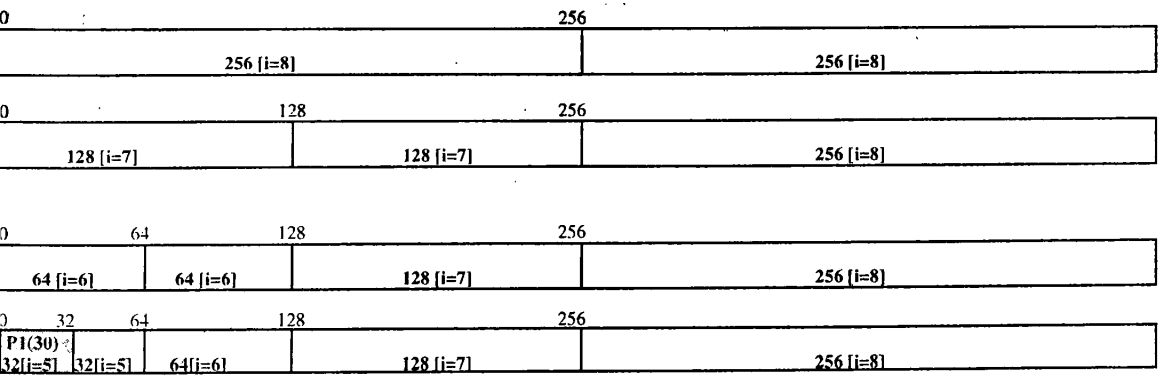
(i) Initially all the free lists are empty except the 9-list which contains a block that starts at address 0.



Free lists : 3→N, 4→N, 5→N, 6→N, 7→N, 8→N, 9→0

(ii) Allocate 30K for P1.

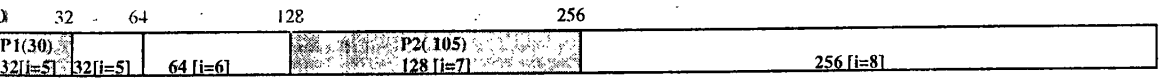
$2^4 < 30 <= 2^5$ so p=5. We look at the 5-list, it is empty therefore we'll look at the 6-list so that we can split a 6-block into two 5-block buddies, but 6-list is also empty. The 7-list and 8-list are also empty. There is a block in the 9-list so it will be splitted into two 8-block buddies at addresses 0 and 256. The 8-block at 256 is put into the 8-list and the other buddy is again spiltted into two 7-block buddies. The 7-block at address 128 is put into 7-list and its buddy at address 0 is again spiltted to give two 6-block buddies. The 6-block at address 64 is put into 6-list and its buddy is splitted into two 5-block buddies. The 5-block at address 32 is put into 5-list and its buddy at address 0 is allocated for P1.



Free lists : 3→N, 4→N, 5→32, 6→64, 7→128, 8→256, 9→N

(iii) Allocate 105K for P2

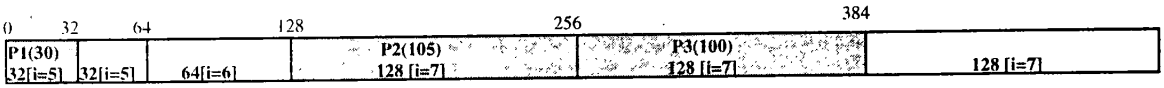
$2^6 < 105 <= 2^7$, so p=7. There is a block in 7-list so we'll allocate this block for P2.



Free lists : 3→N, 4→N, 5→32, 6→64, 7→N, 8→256, 9→N

(iv) Allocate 100K for P3

$2^6 < 100 <= 2^7$, so p=7. The 7-list is empty so we look at the 8-list which has one block. This block is splitted into two 7-block buddies at addresses 256 and 384. The 7-block at address 384 is put into the 7-list and its buddy at 256 is allocated for P3.



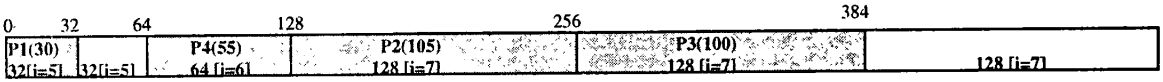
Free lists : 3→N, 4→N, 5→32, 6→64, 7→384, 8→N, 9→N

(v) Allocate 140K

$2^7 < 140 <= 2^8$, so p=8. The 8-list is empty, so we look at the 9-list which is also empty. Therefore this request can't be fulfilled.

(vi) Allocate 55K for P4

$2^5 < 55 <= 2^6$, so p=6. There is a block in 6-list so it is allocated for P4.



Free lists : 3→N, 4→N, 5→32, 6→N, 7→384, 8→N, 9→N

(vii) Allocate 80K for P5

$2^6 < 80 <= 2^7$, so p=7. There is a block in 7-list so it is allocated for P5.

Free lists : 3→N, 4→N, 5→32, 6→N, 7→N, 8→N, 9→N

(viii) Allocate 28K for P6

$2^4 < 28 <= 2^5$, so p=5. There is a block in 5-list so it is allocated for P6.

Free lists : 3→N, 4→N, 5→N, 6→N, 7→N, 8→N, 9→N

When a free block is returned, we need to compute the address of its buddy. A block can have either a left buddy or a right buddy. If the size of block is 2^i and it is situated at address $k * 2^i$, then the block has a right buddy if k is even while it has a left buddy if k is odd. If the block has left buddy then the address of left buddy would be $(k-1) * 2^i$, and if the block has right buddy then the address of right buddy would be $(k+1) * 2^i$. For example a block of size 2^7 situated at address $256(2* 2^7)$ will have a right buddy situated at $384(3*2^7)$. A block of size 2^5 situated at address $32(1 *2^5)$ will have a left buddy situated at address $0(0*2^5)$

The address of the left or right buddy of a block of size 2^i can also be determined by complementing the i^{th} bit in the address of the block(if we count the least significant bit as the 0^{th} bit). If the i^{th} bit in the address of a block of size 2^i is 1, then the block has a left buddy otherwise it has a right buddy.

For example suppose we have block of size 2^5 at address 384,

110 000 000

5^{th} bit is 0, so it will have a right buddy whose address can be obtained by complementing the 5^{th} bit 110 100 000(416).

If we have a block of size 2^7 at address 384,

110 000 000

7^{th} bit is 1, so it will have a left buddy whose address can be obtained by complementing the 7^{th} bit 100 000 000(256).

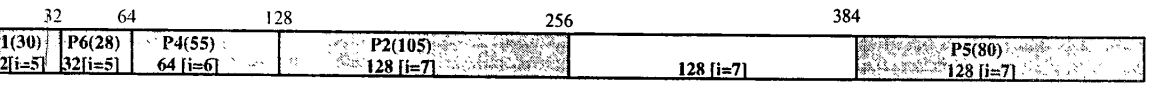
The address of buddy of a block can be obtained by-

ddress of buddy of block B = (Address of block B) xor (Size of block B)

low let us continue from the figure of step(viii) and start freeing the blocks.

x) Free P3

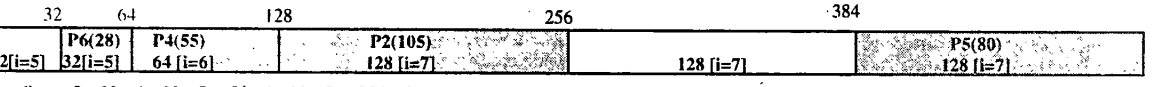
he 7-block at 256 is now free, its buddy at 384 is not free. So the 7-block at 256 is put into the 7-list.



ee lists : 3→N, 4→N, 5→N, 6→N, 7→ 256, 8→N, 9→N

x) Free P1

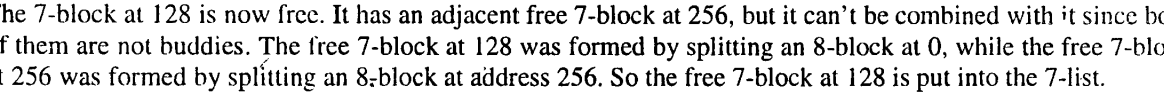
he 5-block at 0 is now free, its buddy at 32 is not free. So the 5-block at 0 is put into the 5-list.



ee lists : 3→N, 4→N, 5→ 0, 6→N, 7→ 256, 8→N, 9→N

xi) Free P2

he 7-block at 128 is now free. It has an adjacent free 7-block at 256, but it can't be combined with it since both f them are not buddies. The free 7-block at 128 was formed by splitting an 8-block at 0, while the free 7-block t 256 was formed by splitting an 8-block at address 256. So the free 7-block at 128 is put into the 7-list.

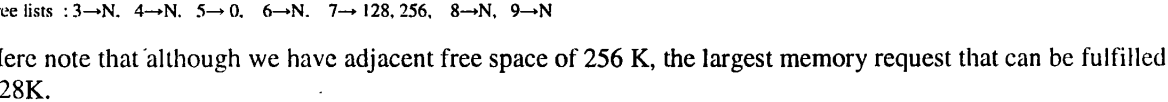


ee lists : 3→N, 4→N, 5→ 0, 6→N, 7→ 128, 256, 8→N, 9→N

lere note that although we have adjacent free space of 256 K, the largest memory request that can be fulfilled is 28K.

xii) Free P6

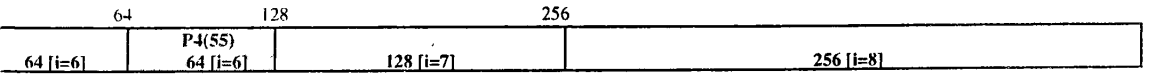
he 5-block at 32 is now free; its buddy is at 0 which is also free so we combine both these buddies to form a 6-lock at 0. Now the buddy of this 6-block at 0 is the 6-block at 64 which is not free, so the free 6-block at 0 is ut into the 6-list.



ee lists : 3→N, 4→N, 5→N, 6→ 0, 7→ 128, 256, 8→N, 9→N

iii) Free P5

he 7-block at 384 is now free and its buddy at 256 is also free, so both of them are combined to form an 8-ock at 256. The buddy of this 8-block at 256 is 8-block at 0, but it is currently not available as it has been litted into smaller blocks. So the free 8-block at 256 is put into the 8-list.

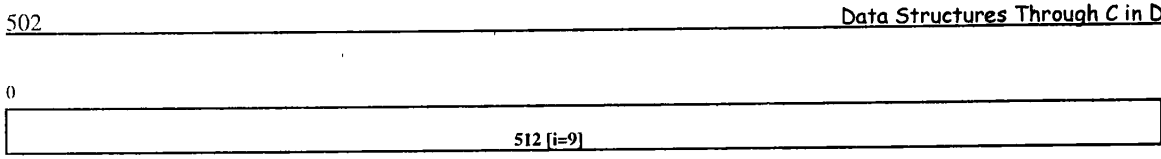


e lists : 3→N, 4→N, 5→N, 6→ 0, 7→ 128, 8→256, 9→N

iv) Free P4

ne 6-block at 64 is now free, its buddy is at 0 and it is also free so both of them are combined to form a free 7-ock at 0. The buddy of this 7-block at 0 is a 7-block at 128 which is also free so both of these are combined to rm a free 8-block at 0. Now the buddy of 8-block at 0 is an

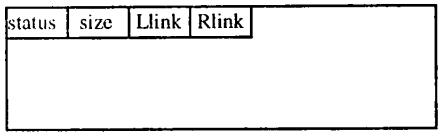
block at 256 which is free so both are combined to form a 9-block at 0. This 9-block is put into the 9-list.



Now let us see what information needs to be stored in each block for the implementation of a buddy system. The four fields which are included in all blocks are-

(i) status flag, which denotes whether the block is free or not
(ii) size, which denotes the size of the block. Instead of storing the exact size we can store the power of 2, e.g. for a block of size 128 we can store 7.
(iii) a left link which points to the previous block on the free list.
(iv) a right link which points to the next block on the free list.

The last two fields are used only when the block is free.



To find whether the buddy of a block B is free or not, first we will calculate the address of the buddy. After that we'll see the size of block located at that address, if the size is not equal to the size of the block B, it means that the buddy of B has been split into smaller blocks. If the size of the block at the calculated address is same as that of B, then we can check the status flag of the buddy to find whether it is free or not.

For example in step (xi), a 7-block located at address 128 is freed. It can be calculated that this block has a left buddy situated at address 0. Now the size field of block at address 0 is checked. It is a 5-block, so this means that the buddy of the block that we have freed just now is not available and it has been splitted into smaller blocks.

In step (ix) a 7-block at address 256 is freed. It can be calculated that this block has a right buddy situated at address 384. Now the size field of the block at address 384 is checked and it is found to be a 7-block, this means the buddy of the block is not splitted. Now the status flag of the buddy is checked which shows that the buddy is not free and so the two buddies cannot be combined.

In step (xiii) a 7-block at 384 is freed. It can be calculated that this block has a left buddy situated at address 256. Now the size field of the block at address 256 is checked and it is found to be a 7-block, this means the buddy of the block is not splitted. Now the status flag of the buddy is checked which shows that the buddy is free and so the two buddies are combined.

For deletion of the buddy from the free list we need address of its predecessor. We are not reaching the buddy by traversing the free list, but we get its address by calculation i.e. we are directly reaching to any position in list. So we need to maintain the free lists as double linked list so that we can get the address of predecessor of a block while removing it from the list.

In binary buddy system, lot of space is wasted inside the allocated blocks. Only memory requests which are in power of 2 result in no waste, while any other size of memory requested will result in wastage of space. Lot of memory is wasted if the size of memory requested is just a bit larger than the smaller block and is very less than the larger block. For example if we need memory of size 130, we will have to allocate a block of size 256 thus wasting 126 memory locations. Hence internal fragmentation is a main disadvantage of binary buddy system. External fragmentation is also present in buddy systems as there can be many adjacent free blocks which can't be combined because they are not buddies.

10.5.2 Fibonacci Buddy System

The Fibonacci sequence can be defined recursively as-

$f_0 = f_1 = 1$
$f_i = f_{i-1} + f_{i-2}$ for $i \geq 2$

The numbers in fibonacci sequence are 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,... Each element is the sum of revious two elements. The Fibonacci buddy system uses these Fibonacci numbers as permitted block sizes. Here an i-block is a block whose size is equal to the i^{th} Fibonacci number. For example a 7-block is a block of ize 21. For the implementation of Fibonacci buddy system, an array of Fibonacci numbers needs to be stored o that the i^{th} Fibonacci number can be found easily.

In binary buddy system, when an i-block splits we get two same size (i-1)-blocks. In Fibonacci system we vill get buddies of different sizes. When an i-block splits, we get one (i-1)-block and one (i-2)-block. For xample if we split a 10-block (size 89), we get a 9-block(size 55) and a 8-block(size 34). In binary buddy ystem we could easily compute the address of the buddy of a block B, from the address and size of block B. Here this computation is not simple as the buddies are of different sizes. We need to store some additional nformation in each block for obtaining the address of its buddy. In each block a left buddy count(LBC) field is ntroduced. Initially the left buddy count of the whole block is 0. This count is changed when a block is split or wo buddies are combined.

Splitting:

LBC of left buddy = LBC of parent block + 1

LBC of right buddy = 0

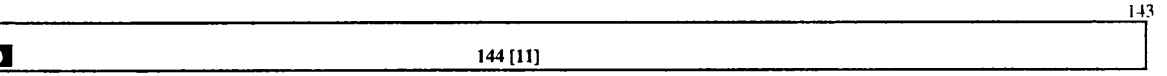
Combining:

LBC of combined block = LBC of left buddy – 1

If the LBC of a block is 0, this means that it is a right buddy. For example if a block of size 34 has LBC 0, his means that it is a right buddy formed by splitting a block of size 89, and the left buddy is of size 55. If the LBC is 1 or more than 1, this means that it is a left buddy.

Suppose we have 144 K of memory, and the smallest size of block that can be allocated is taken as fib(5) = K. So we will have to maintain 7 free lists viz. 5-list, 6-list, 7-list, 8-list, 9-list,10-list,11-list.

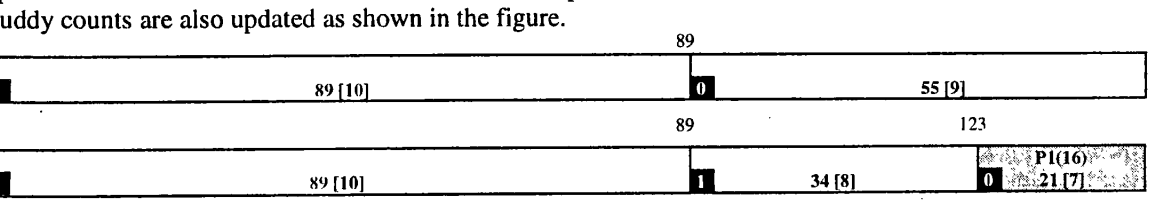
i) Initially all the free lists are empty except the 11-list which contains a block that starts at address 0. The LBC f this block is 0.



Free lists : 5→N, 6→N, 7→N, 8→N, 9→N, 10→N, 11→0

ii) Allocate 16K for P1

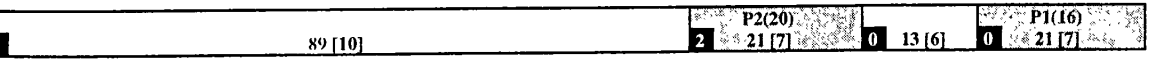
ib(6) < 16 <= fib(7), so we need to allocate a 7-block. The 7-list, 8-list, 9-list, 10-list are all empty so the 11-lock in the 11-list is splitted into a 10-block and a 9-block. The 10-block is put on free list while the 9-block is plit into an 8-block and a 7-block. The 8-block is put on the free list and the 7-block is allocated. The left uddy counts are also updated as shown in the figure.



ee lists : 5→N, 6→N, 7→N, 8→89, 9→N, 10→0, 11→N

ii) Allocate 20K for P2

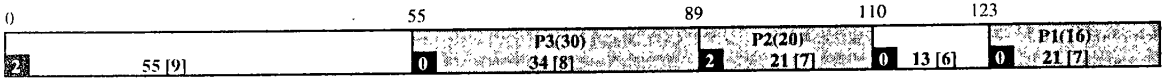
b(6) < 20 <= fib(7), so we need to allocate a 7-block. The 8-block at address 89 is splitted.



ee lists : 5→N, 6→110, 7→N, 8→N, 9→N, 10→0, 11→N

(iv) Allocate 30K for P3

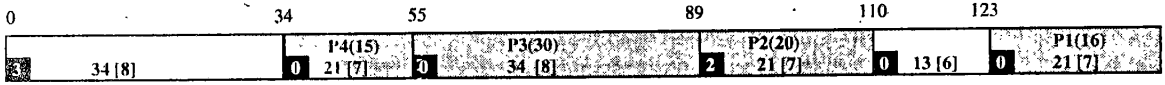
fib(7) < 30 <= fib(8), so we need to allocate an 8-block. The 10-block at address 0 is splitted.



Free lists : 5→N, 6→110, 7→N, 8→N, 9→0, 10→N, 11→N

(v) Allocate 15K for P4

fib(6) < 15 <= fib(7), so we need to allocate a 7-block. The 9-block at address 0 is splitted.



Free lists : 5→N, 6→110, 7→N, 8→0, 9→N, 10→N, 11→N

(vi) Allocate 20K for P5

fib(6) < 20 <= fib(7), so we need to allocate a 7-block. The 8-block at address 0 is splitted.

Free lists : 5→N, 6→21, 110, 7→N, 8→N, 9→N, 10→N, 11→N

(vii) Allocate 10K for P6, 8K for P7

fib(5) < 10 <= fib(6), fib(5) < 8 <= fib(6) so we need to allocate two 6-blocks.

Free lists : 5→N, 6→ N, 7→N, 8→N, 9→N, 10→N, 11→N

(viii) Free P7

The LBC of block P7 is 0 which means that it is a right buddy. Its left buddy would be a 7-block situated at address 0(21 – 21). There is a 7-block at address 0 but it is not free so no combination takes place.

Free lists : 5→N, 6→ 21, 7→N, 8→N, 9→N, 10→N, 11→N

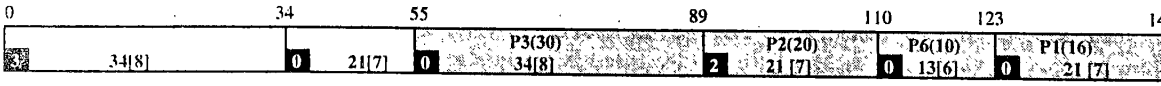
(ix) Free P4

The LBC of block P4 is 0 which means that it is a right buddy. Its left buddy would be an 8-block situated at address 0(34 – 34). The block at address 0 is not an 8-block which means that the left buddy has been splitted.

Free lists : 5→N, 6→ 21, 7→34, 8→N, 9→N, 10→N, 11→N

(x) Free P5

The LBC of P5 is not zero which means that it is left buddy. Its right buddy would be a 6-block situated at address 21(0 + 21). The block at address 21 is a 6-block and is free so combination takes place.



Now the newly combined node is an 8-block at address 0. Its LBC is 3 indicating that it is a left buddy. Its right buddy would be a 7-block at address 34(0+34). The right buddy is free so again a combination takes place.

ee lists : 5→N. 6→ N. 7→N. 8→N. 9→0. 10→N. 11→N

he LBC of block P2 is not 0 which means that it is a left buddy. Its right buddy would be a 6-block at address 10(89+21). The right buddy exists but is not free.

ee lists : 5→N. 6→ N. 7→89. 8→N. 9→0. 10→N. 11→N

xii) Free P1

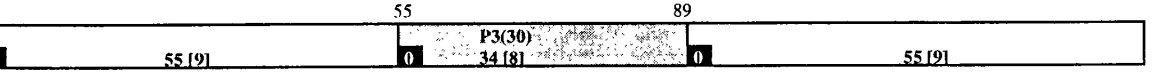
he LBC is zero indicating that it is a right buddy. Its left buddy would be an 8-block at address 89(123- 34). he left buddy is not currently available.

ee lists : 5→N. 6→ N. 7→89.123. 8→N. 9→0. 10→N. 11→N

xiii) Free P6

he LBC is zero indicating that it is a right buddy. Its left buddy would be a 7-block at address 89(110 -21). The ft buddy is free so they are combined.

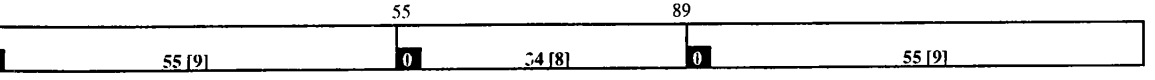
he LBC of combined block is 1, hence it is a left buddy. Its right buddy is a 7-block at address 123(89+34), nd it is free so they are combined.

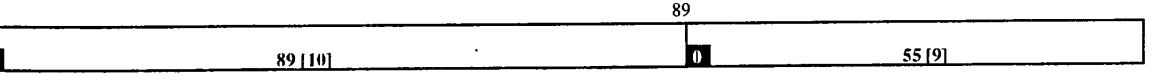


ee lists : 5→N. 6→ N. 7→N. 8→N. 9→0,89. 10→N. 11→N

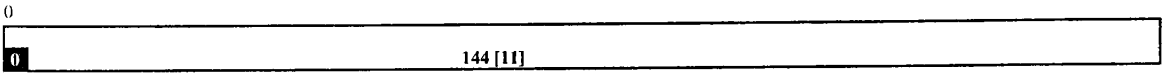
xiv) Free P3·

he LBC is zero, so it is right buddy. Its left buddy is a 9-block at address 0(55-55), and it is free so they are ombined.





he LBC of combined block is 1, hence it is a left buddy. Its right buddy is a 9-block at address 89(0+89), and it free so they are combined.

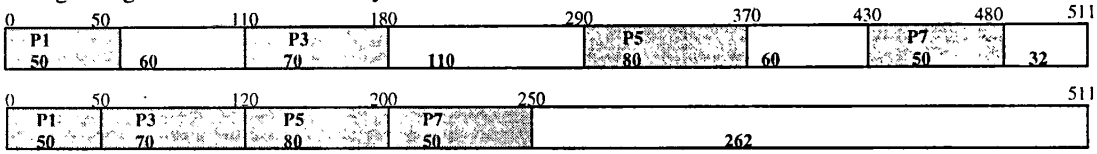


Free lists : 5→N. 6→ N. 7→N. 8→N. 9→N. 10→N. 11→0

In Fibonacci buddy system, internal fragmentation is reduced since there is a larger variety in the sizes of free blocks.

10.6 Compaction

After repeated allocation and dealloctaion of blocks, the memory becomes fragmented. Compaction is a technique that joins the non contiguous free memory blocks to form one large block so that the total free memory becomes contiguous. All the memory blocks that are in use are moved towards the beginning of the memory i.e. these blocks are copied into sequential locations in the lower portion of the memory. For example in the following figure, blocks allocated for programs P1, P3, P5, P7 are copied in locations starting from 0 and a single large block of free memory is created.



When compaction is performed, all the user programs come to a halt. A problem can arise if any of the used blocks that are copied contain a pointer value. For example suppose inside block P5, the location 350 contains address 310. After compaction the block P5 is moved from location 290 to location 120, so now the pointer value 310 stored inside P5 should change to 140. So after compaction the pointer values inside blocks should be identified and changed accordingly.

10.7 Garbage collection

Garbage refers to those memory blocks that are allocated but not in use, these objects are dead. The garbage collection technique is used to recognize garbage blocks and automatically free them. Garbage collection is also known as automatic memory management, as the dynamically allocated memory is automatically reclaimed by the garbage collector, and there is no need for the programmer to deallocate it explicitly. The main work of a garbage collector is to differentiate between garbage and non garbage blocks and return the garbage blocks to the free list. The two common approaches of garbage collection are-

(i) Reference counting
(ii) Mark and sweep

10.7.1 Reference counting

Each allocated block contains a reference count which indicates the number of pointers pointing to this block. This count is incremented each time we create or copy a pointer to the block and is decremented each time when a pointer to the block is destroyed. When the reference count of an object becomes zero, it become unreachable and is considered as garbage. This garbage block is immediately made reusable by placing it on the free list.

An advantage of reference counting is that a block of memory is freed as soon as it becomes garbage. A major disadvantage of reference counting is that it cannot handle cyclic references correctly. A cyclic reference occurs when an object references itself indirectly, for example when some block A references block B and block B references block A. The reference count of blocks A and B will never become zero. So reference counting mechanism fails to recognize cyclic data structures as garbage and is not able to free them. Another disadvantage is that reference counts have to be frequently updated thereby increasing processing costs.

10.7.2 Mark and Sweep

The mark and sweep garbage collector is run when the system is very low on memory and it is not possible to allocate any space for user. All the application programs come to a halt temporarily when this garbage collector runs, and resume when all the garbage blocks are reclaimed. This garbage collection takes place in two phases, the first phase is the mark phase in which all the non garbage blocks are marked and second is the sweep phase in which the collector sweeps over the memory and returns all the unmarked(garbage) blocks to the free list.

A root is a program variable which directly points to a block on the heap and the set of all the roots is called the root set. These roots may be local variables on stack frames, register variables, global variables or static variables. A block is live or reachable if it is directly or indirectly accessible by the root set. The directly accessible blocks are those which are pointed to by any root, and the indirectly accessible blocks are those which are pointed to by any pointer from within a live block. Hence all the reachable blocks can be found out by following pointers from the root set.

So the first task that is to be done is to find out the root set. For this all the program variables are scanned and pointers to dynamic memory(heap) are identified as roots. All the blocks that are directly and indirectly referenced by these roots are visited and marked. This is like DFS traversal of a graph and can be implemented recursively. The traversal starts from the set of roots and all the reachable blocks are visited. Whenever a block is visited, its marked field is set to true. So after the first phase all live blocks are marked and garbage blocks are not.

In the sweep phase, the garbage collector sequentially scans all the blocks on the heap and reclaims all the unmarked ones by placing them on the free list. The marked blocks are unmarked for the next run of the garbage collector. There is no movement of blocks.

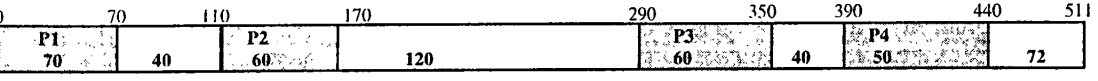
In each memory block, a boolean field is taken to differentiate between the marked and unmarked nodes. This mark field will be true if the block is marked and false if it is unmarked.

A mark and sweep garbage collector can recognize blocks that have already been marked and so there is no problem in the case of cyclic references. Another advantage is that there is no overhead of maintaining reference variables as in the reference count method.

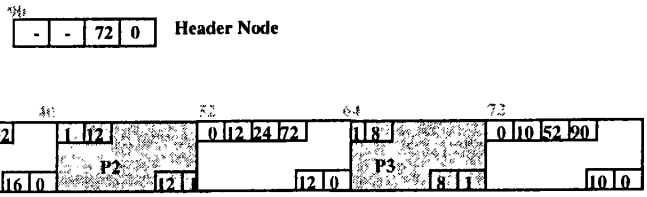
The disadvantage of this method is that it uses a stop-the-world approach i.e. all programs need to stop when garbage collection takes place. This may be undesirable in interactive and real time applications. Another problem named thrashing occurs when most of the memory is being used. In this case the collector is able to reclaim very less memory which is exhausted in a short duration. This causes the garbage collector to be called again, and this time also it reclaims only little space. So the garbage collector is called again and again in this case.

Exercise

1. Consider the situation in the given figure and allocate 50K for a program using-
(i) First Fit method
(ii) Best Fit Method
(iii)Worst Fit method

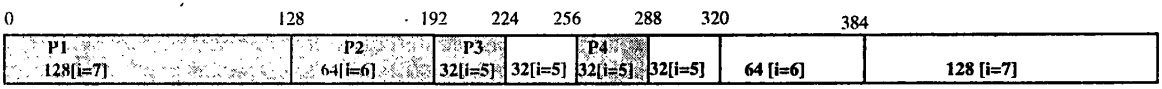


2. Deallocate the memory block P2 using Boundary tag method.



3. Suppose we have 512K of free memory and the smallest block of memory that can be allocated is of size 8K. Use binary buddy system to allocate 100K for P1, 25K for P2, 50K for P3, 60K for P4, 125K for P5.

4. The following memory blocks were allocated using binary buddy system. Deallocate the block P4.



5. Suppose we have 144K of free memory and the smallest block of memory that can be allocated is of size 8K. Use fibonacci buddy system to allocate 20K for P1, 30K for P2, 15K for P3, 35K for P4.

6. The following memory blocks were allocated using fibonacci buddy system. Deallocate the block P3.

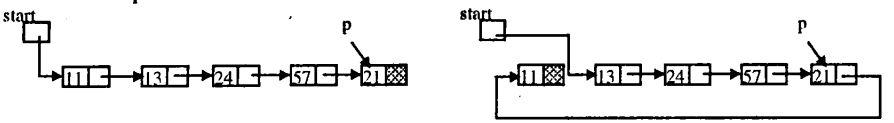
SOLUTIONS

Chapter 2
Arrays, Pointers and Structures

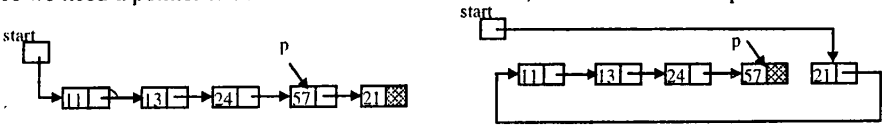
. Error, the size of an array should be a constant expression. **2.** j = 5 **3.** sum = 12 **4.** sum = 60
. a = 4, b = 6
arr1[0] = 6, arr1[4] = 10
arr2[0] = 1, arr2[4] = 5
. 25 25 30 30 35 35 40 40 45 45
. Error, since arr is a constant pointer and it can't be changed.
. 26 31 36 41 46
By (*p)++ we increment the value pointed to by p, and by p++ we increment the pointer.
. 25 30 35 40 55 **10.** 35 55 65 **11.** 90 85 70 65 60 55 40 35 30 25
2. 90 85 70 65 60 55 40 35 30 25 **13.** 20 20 10 **14.** a = 2, b=2 **15.** 4 5 6 7
6. Marks = 80 Grade = A Marks = 80 Grade = A
7. Error - The structure definition is written inside main( ), so it can't be accessed by function func( ). It should be written before main( ) so that it is accessible to all functions.
8. 12 **19.** 13

Chapter 3
Linked List

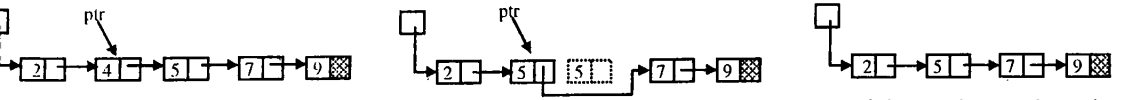
. Traverse the list L, and keep on inserting the nodes one by one at the beginning of the new list.
. We did this in a single pass of bubble sort; here we will do the swap unconditionally. After swapping two nodes we will eave them and move to the next node.
. Unlike single linked list, here we will not need a pointer to the previous node
. While swapping through info, we will need a pointer to the last node. While swapping through links, we will need a ointer to the second last node.
. Compare adjacent elements and if the first one is greater than second, swap them. This is the technique we used in a ingle pass of bubble sort.
0. Compare the first element with all other elements starting from the second one, and if any element is smaller than the rst element, swap them. This is the technique we used in a single pass of selection sort.
3. Here we need a pointer to the last node of the list.



4. Here we need a pointer to the second last node of the list, because we have to put NULL in its link.



6. For deleting a node we need a pointer to the previous node, so that we can change link part of previous node and make it oint to the node which is after the node to be deleted. Here we are given only a pointer to the node to be deleted, and since ıe list is single linked list we can't get the pointer to previous node. We don't have the start pointer so we can't traverse the st and get the pointer to previous node.
Ve copy the data from the next node into the node to be deleted and then delete the next node.



he last node cannot be deleted if only a pointer to it is given, because in that case we need the previous node so that its link ın be set to NULL.

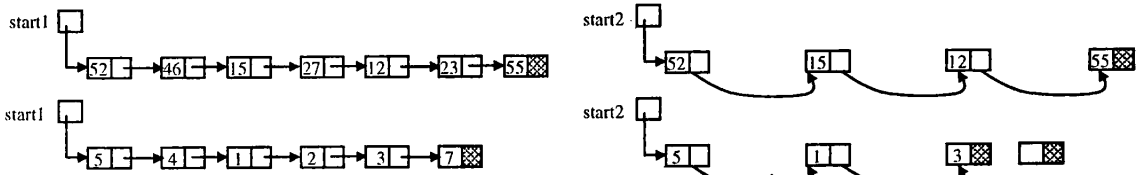
17. Inserting a node after a node pointed by p is simple. For inserting a node before a node pointed by p we need the pointer to previous node also. But we have no other information except the pointer p. So we can use the trick of copying info as we did in exercise 16.

18. After freeing the pointer p, the expression `p->link` is meaningless. So we need to store `p->link` in some variable before freeing the pointer p. At last the pointer start should be assigned NULL.

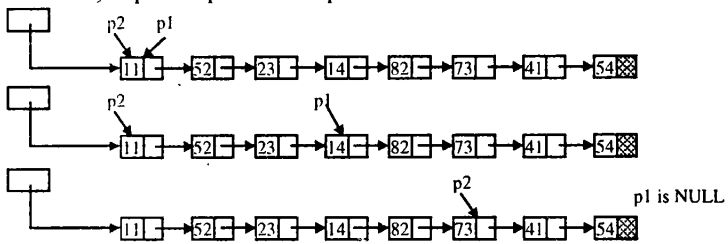
19. Since the list is sorted, the duplicate elements will be adjacent. We can start traversing from the beginning and compare the data of adjacent nodes. If the data is same we can delete the next node and continue. Before deleting any node we need to store the address of its next node.

20. In an unsorted list, the duplicate elements need not be adjacent. So here, either we can sort the list and then delete duplicates as in E19 or we can use two nested loops to compare the elements of both lists.

25.

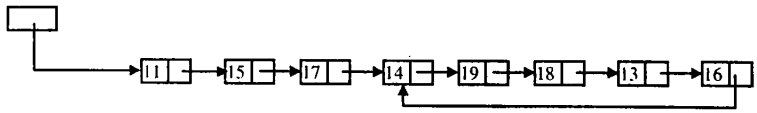


26. Take two pointers and initially point them to the first node. Move the first pointer n times, now move both the pointers simultaneously and when the first pointer will become NULL, the second pointer will be pointing to the n^{th}-last node. For example to find 3^{rd} last node in the list given below, first move pointer p1 three times, then move both p1 and p2 till p1 becomes NULL. The pointer p2 will then point to the 3^{rd} last node.



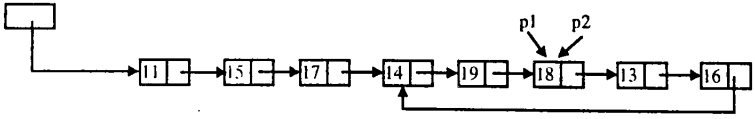
27. A simple solution to find a cycle in linked list is to have a visited flag in every node of the linked list. The list is traversed from the beginning and as each node is visited, its visited flag is set to true. If we reach a node whose visited flag is true then there is a cycle in the list.

Another solution which is efficient and in which we need not make any changes in the nodes of the list, is based on Tortoise and Hare algorithm. Take two pointers and initially point them to the first node. Move the first pointer one node at a time and move the second pointer two nodes at a time. If the list is not NULL terminated i.e. it contains a cycle then after some time both the pointers will enter the cycle and will definitely meet at some node. The fast and slow pointer can meet only if there is a cycle in the node or the whole list is circular. This algorithm is called tortoise and hare algorithm because we have used a slow pointer(tortoise) and a fast pointer(hare). This algorithm is also known as Floyd's cycle detection algorithm.



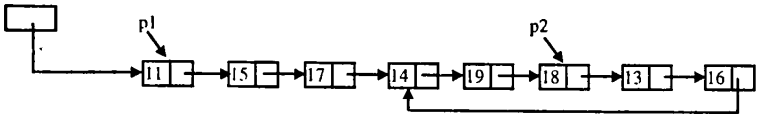
`p1->11, p2->11` `p1->15,p2->17` `p1->17,p2->19` `p1->14,p2->13` `p1->19,p2->14`
`p1->18, p2->18`

Both pointers p1 and p2 meet at the node 18, so there is a cycle in the list.



To count the number of nodes in the cycle, fix the pointer p1 at node 18 and move the pointer p2 one node at a time. Stop the pointer p2 when it meets pointer p1. The pointer p2 visits a full cycle and the number of times it moves is equal to the number of nodes in the cycle(5 nodes).

To count the remaining nodes in the list, make the pointer p1 point to the first node. The pointer p2 points to node 18. Now move both pointers till they meet.



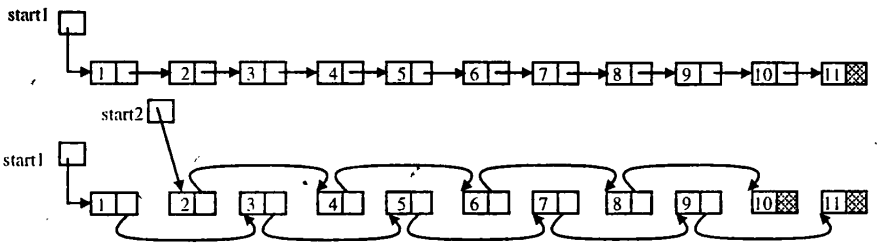

```
p1->11, p2->18      p1->15,p2->13   p1->17,p2->16   p1->14,p2->14
```

The two pointers meet at node 14, so it is the node that causes the cycle. The node 14 is pointed by nodes 17 and 16. The number of times the pointers move gives us the number of nodes which are not in the cycle(3 nodes). So the total number of nodes in the list is 8(5+3). Now we know the length of the list so we can remove the cycle by entering NULL in the link of last node(8^{th} node).

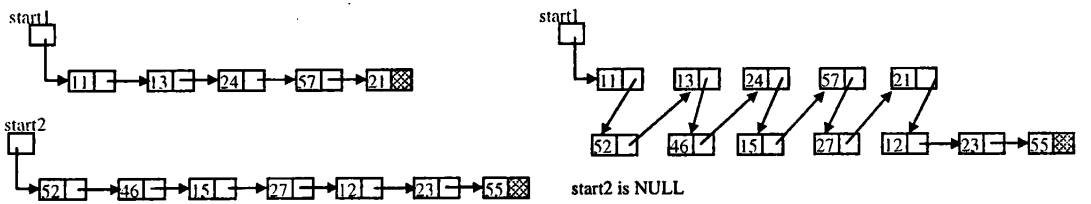
28. We can use the tortoise and hare algorithm. When the fast pointer becomes NULL, the slow pointer will be at the middle of the list.

29. Reach the middle of the list as in E28, and then split the list there.

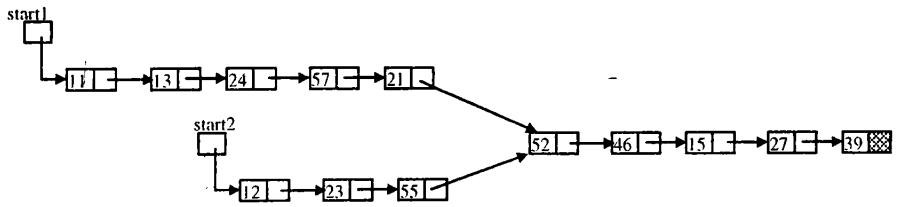
31.



32.



33.



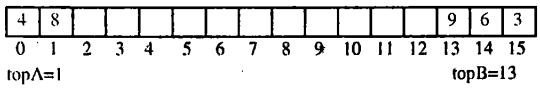
Find the length of both lists. Suppose d is the difference of lengths of both lists. Take two pointers pointing to beginning of the lists. Move the pointer which points to the longer list d times. Now move both pointers simultaneously and compare them. If at any point both the pointers are equal then we get a merge node. If we reach the end of any of the lists without getting the merge node, we can conclude that the lists don't intersect at any point. In the example, the merge point is the node containing data 52.

36. First find the middle of the list as in E28. Split the list into two halves, reverse the second half of the list and compare the two halves. After comparing reverse the second half and join the two halves to get the original list. There will be different cases for odd and even elements.

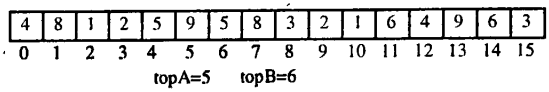
Chapter 4
Stack and Queue

4. First stack will start from 0^{th} position and second stack will start from last position of array. The overflow will occur when the top of the two stacks cross

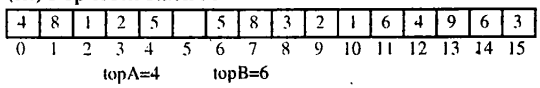
(i) Push 4, 8 on stack A, Push 3,6,9 on stack B



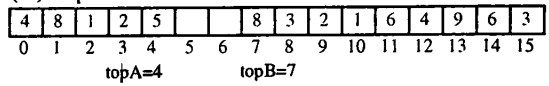
(ii) Push 1,2,5,9 on stackA, Push 4,6,1,2,3,8,5 on stack B



(iii) Pop from stack A



(iv) Pop from stack B

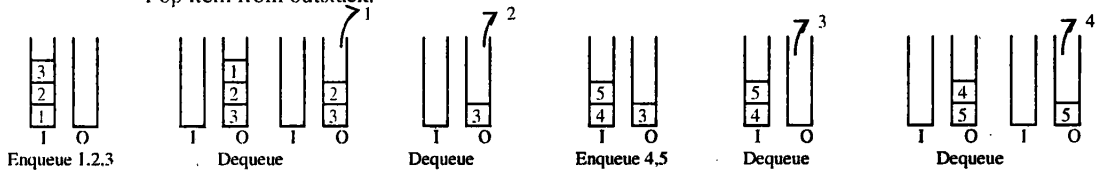


5. First queue will start from 0^{th} position and second queue will start from last position of array.

6. Enqueue - Push item on instack

Dequeue - If outstack is empty then move all the items from instack to outstack.

Pop item from outstack.



This way we can get the FIFO behavior using two stacks.

7. Push – Enqueue in Q1

Pop – Delete all items from Q1 except the last and insert in Q2. The last item left is the item to be popped. Now insert all items back into Q1.

8. Push - Enqueue the item

Pop - Delete all the items one by one and insert then at the end one by one except the last one which is the item popped

(i) Push 1,2,3,4,5

Queue - 1,2,3,4,5

(ii) Pop

Queue - 2,3,4,5,1

Queue - 3,4,5,1,2

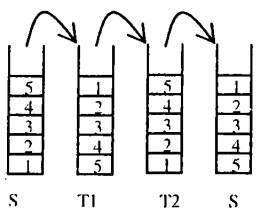
Queue - 4,5,1,2,3

Queue - 5,1,2,3,4

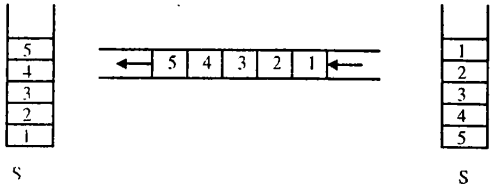
Now 5 is deleted which is the item popped

Queue - 1,2,3,4

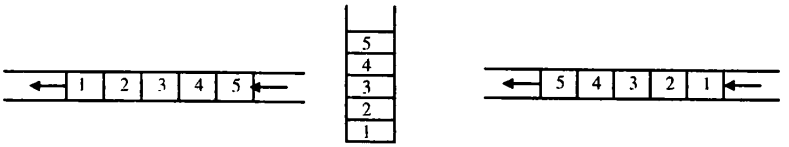
9.



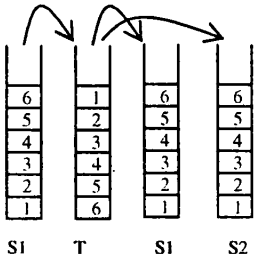
10.



11.



12.



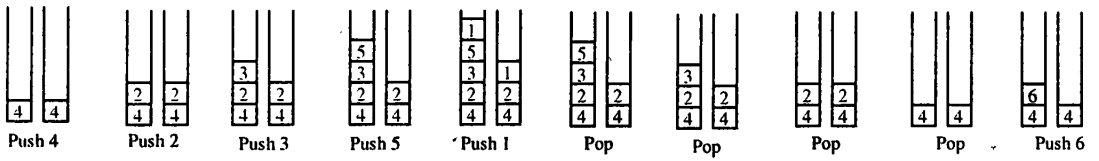
13. Delete the elements from Q1 one by one and insert them both in Q1 and Q2.

Q1- 1 2,3,4,5	Q1- 2,3,4,5,1	Q1- 3,4,5,1,2	Q1- 4,5,1,2,3	Q1- 5,1,2,3,4	Q1- 1,2,3,4,5
Q2-	Q2- 1	Q2- 1,2	Q2- 1,2,3	Q2- 1,2,3,4	Q2- 1,2,3,4,5

14. Take two stacks, one to store all the elements and other to store the minimum.

Push – Push the item on the main stack. Push on the minimum stack only if the item to be pushed is less than or equal to the value at the top of minimum stack.

Pop – Pop from the main stack. Pop from the minimum stack only if the value popped from main stack is equal to that on the top of minimum stack.



15. Keep on dividing the number by 2 and pushing the remainders on a stack till the number is reduced to 0. Now pop all the numbers from the stack and display them

101

101/2 q=50, r=1 stack=1
50/2, q=25,r=0 stack=1,0
25/2, q=12, r=1 stack=1,0,1
12/2,q=6,r=0 stack=1,0,1,0
6/2,q=3,r=0 stack=1,0,1,0,0
3/2,q=1,r=1 stack=1,0,1,0,0,1
1/2, q=0,r=1 stack=1,0,1,0,0,1,1
Binary = 1100101

16. Find the prime factors iteratively and push them in a stack. Pop elements from the stack and display them.

17. Scan the infix from right to left. Whenever an operator comes, pop the operators which have priority greater than(not equal to) the priority of the symbol operator. Rest of the procedure is same as in postfix.

18. Postfix (i) AB+CD+* (ii) ACD-%BE*+ (iii) ABC+D*E/- (iv) HJK+^I*S%
Prefix (i) *+AB+CD (ii) +%A-CD*BE (iii) -A/*+BCDE (iv) %*^H+JKIS

19. (i) -/+5342 Value : 0 (ii) ++/63*3^221 Value : 15 (iii) +-*/+8223/213 Value : 16

20. Scan from left to right, when you get an operator; place it before the 2 operands that precede it. For example-
(i) ABC*-DE-F*G/H/+
A[*BC]-DE-F*G/H/+
[-A*BC]DE-F*G/H/+
[-A*BC][-DE]F*G/H/+
[-A*BC][*-DEF]G/H/+
[-A*BC][/*-DEFG]H/+
[-A*BC][//*-DEFGH]+
+-A*BC//*-DEFGH

(ii) Prefix: -*/+ABCD^EF (iii) Prefix : +-A^BC/*DE+FG

21. Scan from right to left, when you get 2 consecutive operands after an operator, take the operator and place it after the two operands. For example-

(i) +-+A/BC*DEF
+-+A[BC/]*DEF
+-[ABC/+]*DEF
+-[ABC/+][DE*]F
+[ABC/+DE*-]F
ABC/+DE*-F+

(ii) Postfix : ABCDE^*/+FGH^*- (iii) Postfix : ABCD^/+EF*GH+/-

22. (i) 14 (ii) 9 (iii) 17

Chapter 5
Recursion

1. 33 33. Both functions return the sum of all numbers from a to b.

2. 1033, Function returns the sum of all numbers from a to b added to 1000.

3. 40, func( ) returns the sum of all numbers from 6 to 10. In func1( ), we have infinite recursion as the terminating condition is never true.

4. func(4,8) returns 30, but in func(3,8) we have the infinite recursion as the terminating condition is never true.

5. 18 17 16 15 14 13 12 11 10
10 11 12 13 14 15 16 17 18

6. 10 11 12 13 14 15 16 17 18
18 17 16 15 14 13 12 11 10

7. 24 0 0 , func( ) returns the product of a and b. **8.** 5, count( ) returns the number of digits in n.

9. 23 , func( ) returns the sum of digits in n. **10.** 3, count( ) returns the number of times digit d occurs in number n.

11. 4 , func( ) returns the total number of even elements in array arr.

12. 28 , func( ) returns the sum of all elements of array arr.

13. func( ) returns the number of times character c occurs in string str.

14. func1( ) prints - func2( ) prints-

```
****                    *
***                     **
**                      ***
*                       ****
```

15. 1, func( ) returns the smallest element of the array.

16. 2, func( ) returns the smallest element of the array. The array arr[low], ...arr[high] is divided into two parts arr[low], ...arr[mid] and arr[mid+1], ...arr[high] and the smallest elements of both parts is computed recursively. The smaller of these two elements is returned as the smallest element of the array. If the size of array is even then the size of both parts is same, and if the size of array is odd then the size of first part is one more than the size of second part. The terminating condition for recursion is when the size of part becomes equal to one i.e. `low==high`.

23. Proceed as in E16.

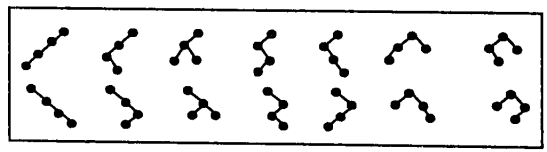
30.

$$a^n = \begin{cases} 1 & \text{if } n=0 \\ (a*a)^{n/2} & \text{if n is even} \\ a * (a* a)^{(n-1)/2} & \text{if n is odd} \end{cases}$$

Chapter 6
Trees

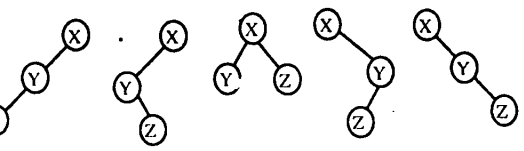
1.

Non similar Binary trees having 3 nodes

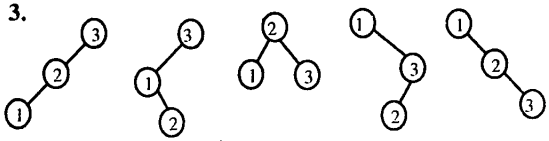


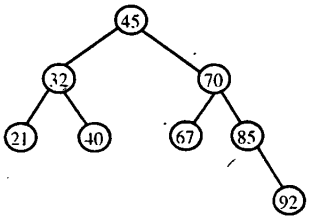
Non Similar Binary trees having 4 nodes

here are n nodes then the number of possible non similar binary trees is $^{2n}C_n * 1/(n+1)$
r 3 nodes [6! / (3! * 3!)] * [1/4] = 5
r 4 nodes [8! / (4! * 4!)] * [1/5] = 14
we have to find out the total number of different binary trees with n different keys, then we can multiply the above value- n!.
r example suppose we have to find number of possible binary trees of 3 nodes having key values 1,2,3. We have 5 ferent structures of possible binary trees as shown in the figure, and in each structure the values 1,2,3 can be arranged in ways. So the total number of different binary trees will be 30.

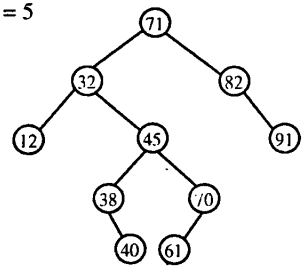


3.



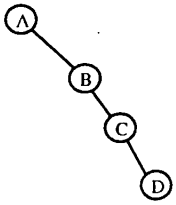


5. Height = 5



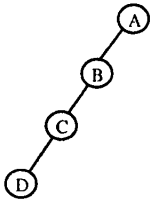
If data is inserted in sorted order then the height of the binary search tree will be 10.

Inorder traversal of binary search tree is always in ascending order, so arranging the data in ascending order gives us inorder traversal.
Inorder - 6 9 11 12 23 32 45 56 67
From inorder and preorder traversals we can construct the tree and find the postorder traversal.
Postorder - 6 9 11 12 32 56 67 45 23

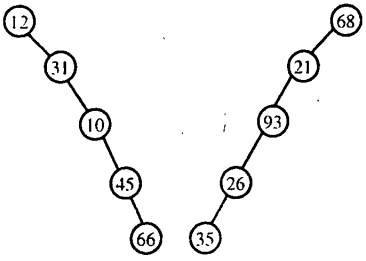


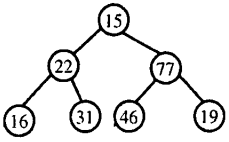
order and preorder are A B C D

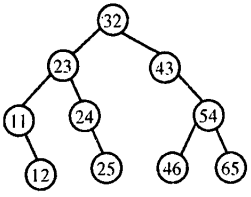
8.



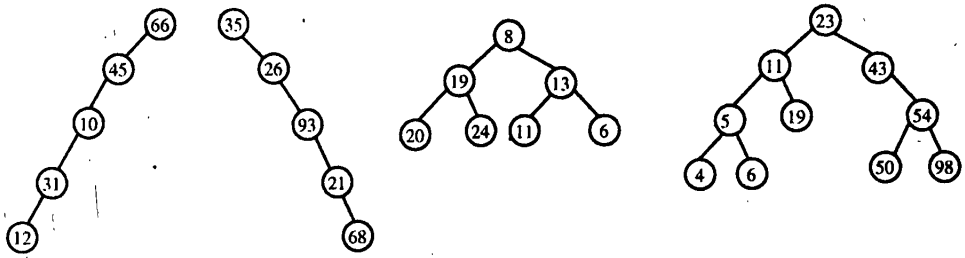
Inorder and postorder are D C B A







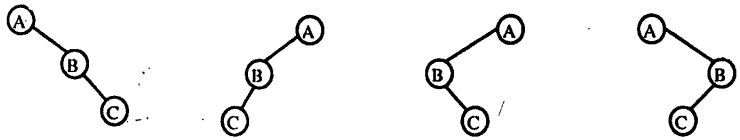
10.



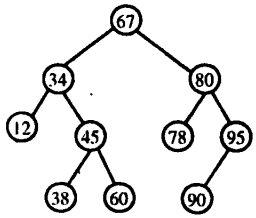
11. (i) (7, 5, 2, 1) (ii) (9, 6, 5, 7, 8) (9, 6, 7, 8, 5) (9, 6, 7, 5, 8)
(iii) (4, 6, 5, 8) (4, 6, 8, 5) (iv) (1, 2, 5, 7)

12. (a) (n-x) (b) y+1

13. For all these trees Preorder = ABC, Postorder = CBA

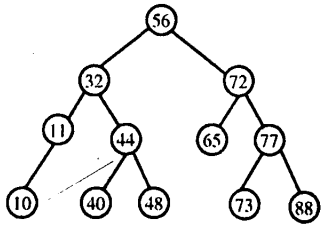


14. 67 is root node, all values less than 67 will be in left Subtree and more than 67 will be in right subtree. Applying this logic for the subtrees also we can create our BST.

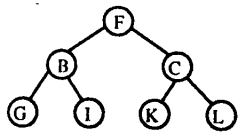


Alternatively, we can construct the tree as in section 6.9.3. The inorder traversal of a BST can be found by putting the data in sorted order.

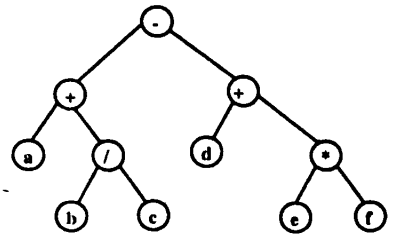
15.



16. In a full binary tree, the left and right subtrees of a node have same number of nodes. F is the root node, and from the remaining 6 nodes three will be in left subtree and three in right subtree.

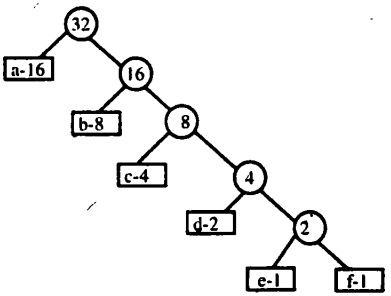


34.



35. After inserting 28 - 50 40 30 28 16 23 20 25
After inserting 43 - 50 43 30 40 16 23 20 25 28
After inserting 11 - 50 43 30 40 16 23 20 25 28 11

36.

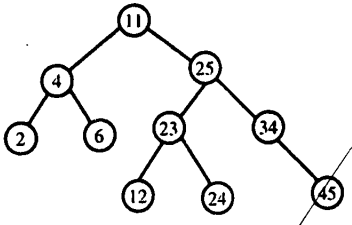


Huffman codes : a-0, b-10, c-110, d-1110, e-11110, f-11111

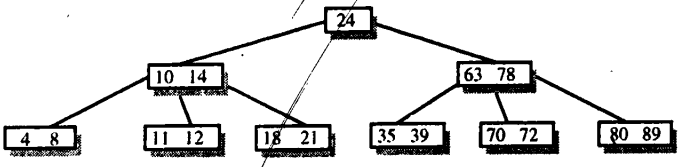
37. The correct function is -

```
int isAVL(struct node *ptr)
{
    int h_l, h_r, diff;
    if(ptr == NULL)
            return 1;
    h_l = height(ptr->lchild);
    h_r = height(ptr->rchild);
    diff = h_l>h_r ? h_l-h_r : h_r-h_l;
    if( diff<=1 &&  isAVL(ptr->lchild) && isAVL(ptr->rchild) )
            return 1;
    return 0;
}
```

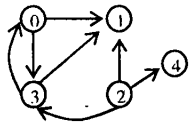
38.



39.



Chapter 7
Graph

1. 

(i) In(0)=1, In(1)=3, In(2)=0, In(3)=2, In(4)=1, Out(0)=2 ,Out(1)=0 , Out(2)=3, Out(3)=2, Out(4)=0
(ii) The path matrix is :

	0	1	2	3	4
0	1	1	0	1	0
1	0	0	0	0	0
2	1	1	0	1	1
3	1	1	0	1	0
4	0	0	0	0	0

(iii) Since all elements of the path matrix are not 1, the graph is not strongly connected.
(iv) Since all the diagonal elements of the path matrix are not 0, the graph is not acyclic.

2. (i) Sum of degrees = 3 + 3 + 3 + 3 = 12 (ii) Sum of degrees = 1 + 0 + 1 + 2 = 4
(iii) Sum of degrees = 3 + 2 + 3 = 8 (iv) Sum of degrees = 1 + 2 + 1 + 4 + 1 + 2 + 1 = 12
Sum of degrees of all the vertices is twice the number of edges. This is because each edge contributes 2 to the sum of degrees. This result is called handshaking Lemma. If in a meeting, people are represented by vertices, and a handshake between two people by an edge, then the total number of hands shaken is twice the total number of handshakes.

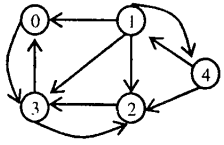
3. Sum of degrees = Sum of degrees of even vertices + Sum of degrees of odd vertices.
The sum of degrees of all vertices is even by handshaking lemma, and the sum of degrees of even vertices will definitely be even. Difference of two even numbers is even, so the sum of degrees of odd vertices will be even. Sum of degrees of odd vertices is a sum of only odd terms, so the sum can be even only if the number of these odd terms is even. So the number of odd vertices in a graph is even.

4. (i) Sum of indegrees = 2 + 1 + 2 + 0 + 1 = 6, Sum of outdegrees = 1 + 1 + 0 + 3 + 1 = 6
(ii) Sum of indegrees = 0+ 2+ 1 = 3, Sum of outdegrees = 2 + 0 + 1 = 3
(iii) Sum of indegrees = 1 + 2 + 1 = 4, Sum of outdegrees = 2 + 1 + 1 = 4
(iv) Sum of indegrees = 2 + 1 + 2 + 3 = 8 , Sum of outdegrees = 2 + 3 + 1 + 2 = 8
Sum of indegrees of all vertices = Sum of outdegrees of all vertices = Number of edges.
This is handshaking lemma for the directed graphs. Each edge contributes 1 to the sum of indegrees and 1 to the sum of outdegrees.

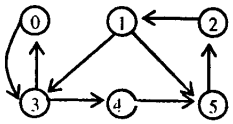
5. The sum of degrees of all vertices of a graph is twice the number of edges. In a regular graph of n vertices having degree d, the sum of degrees of all vertices is n*d.
2c = n*d => c =(n*d)/2
(i) 5 (ii) 6
(iii) (3*5)/2 is not a whole number so this is not possible. In Ex3 we have proved that number of odd vertices should be even but here it is not so.

6. (i) 5, 7, 2 (ii) 1, 2, 5 (iii) 3, 1, 5

7. (i) All vertices are visited by performing a DFS from 0. Now reverse the graph.

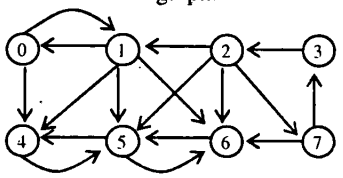


In the reverse graph, all vertices are not visited by performing a DFS from 0. So the graph is not strongly connected.
(ii) All vertices are visited by performing a DFS from 0. Now reverse the graph.



In the reverse graph, all vertices are visited by performing a DFS from 0. So the graph is strongly connected.

8. DFS taking 0 as the start vertex - 0 1 2 3 7 4 5 6
Discovery time and finishing times of vertices are –
Vertex 0 - (1, 10), Vertex 1 - (2, 9), Vertex 2 - (3, 8), Vertex 3 - (4, 7),
Vertex 4 - (11,16), Vertex 5 - (12, 15), Vertex 6 - (13,14), Vertex 7 - (5, 6)
Now reverse the graph.



Start DFS from 4 as it has highest finishing time.
Strongly connected components are - (4, 5, 6) (0,1) (2,3,7)

9. (0, 3, 1) = 9, (0, 3, 4, 5, 2) = 22, (0, 3) = 5, (0, 3, 4) = 14, (0, 3, 4, 5) =16

10. (i) Edges in MST : (0,1), (0,2), (0,3),.....(0,n-1)
Weight of MST = 1 + 2 + 3 +.....(n-1) = n(n-1)/2
(ii) Edges in MST : (0,1), (1,2),(2,3).....(n-2,n-1)
Weight of MST = 1 + 1 + 1 + ...+ 1 = n-1
(iii) Edges in MST : (0,1), (1,2),(2,3).....(n-2,n-1)
Weight of MST = 5 + 5 + 5 +.....+ 5 = 5(n-1)

Chapter 8
Sorting

1. (1 2 3 4 5) (5 4 3 2 1) (2 3 1 5 4)
For all these sets of data the number of comparisons will be 10.

5. 9 12 21 23 32 10 34 45 67 89

8. 12 23_A 42 23_B 6 9 23_C
6 23_A 42 23_B 12 9 23_C
6 9 42 23_B 12 23_A 23_C
6 9 12 23_B 42 23_A 23_C
6 9 12 23_B 42 23_A 23_C
6 9 12 23_B 23_A 42 23_C
6 9 12 23_B 23_A 23_C 42

1. On sorting the data given in E8 by bubble sort we get 6 9 12 23_A 23_B 23_C 42

10. 2 5 5 13 19 20 21 34 89 3

12. (i) Bubble (ii) Selection (iii) Insertion

14. 5 or 8 could be the pivot.

15. 11 21 23 12 12 36 19 35 47 87 72 56

16. The contents of array after each pass of heap sort are-

83 76 82 54 45 21 12 _____ 97
82 76 21 54 45 12 _____ 83 97
76 54 21 12 45 _____ 82 83 97
54 45 21 12 _____ 76 82 83 97
45 12 21 _____ 54 76 82 83 97
21 12 _____ 45 54 76 82 83 97
12 _____ 21 45 54 76 82 83 97

18. Sublist 1 : 34 54 17 12 → 12 17 34 54
Sublist 2 : 21 88 51 → 21 51 88
Sublist 3 : 65 23 72 → 23 65 72
Sublist 4 : 89 76 91 → 76 89 91
Sublist 5 : 11 98 24 → 11 24 98
12 21 23 76 11 17 51 65 89 24 34 88 72 91 98 54

Chapter 9
Searching and Hashing

1. Search 27 (0, 8, 17) (0, 3, 7) - 27 found at position 3
Search 32 (0, 8, 17) (0, 3, 7) (4, 5, 7) (4, 4, 4) (5, 4, 4) - 32 not found in array
Search 61 (0, 8, 17) (9, 13, 17) (9, 10, 12) (9, 9, 9) - 61 found at position 9
Search 97 (0, 8, 17) (9, 13, 17) (14, 15, 17) (16, 16, 17) (17, 17, 17) - 97 found at position 17

2. Squaring the keys and taking 2nd, 4th and 6th digits

116964	**045369**	**186624**	**293764**	**017424**	**582169**	**088804**
194	439	864	974	144	819	884

3. 321 + 982 + 432 = **1735** 213 + 432 + 183 = **828**
343 + 541 + 652 = **1536** 542 + 313 + 753 = **1608**
Addresses are 735, 828, 536, 608

4. 321 + 289 + 432 = **1042** 213 + 234 + 183 = **630**
343 + 145 + 652 = **1140** 542 + 313 + 753 = **1608**
Addresses are 42, 630, 140, 608

5. H(9893)=37, H(2341)=**37**, H(4312)=24, H(7893)=21, H(4531)=51, H(8731)=27, H(3184)=48
H(5421)=45, H(4955)=**27**, H(1496)=**24**
H(9893)=44, H(2341)=63, H(4312)=24, H(7893)=54, H(4531)=42, H(8731)=21, H(3184)=35
H(5421)=61, H(4955)=64, H(1496)=22

6. Linear Probe **7.** Quadratic Probe **8.** Double hashing

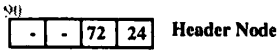
	6. Linear Probe	7. Quadratic Probe	8. Double hashing
[0]	102	102	102
[1]	67	120	120
[2]	120		
[3]	37	37	37
[4]	88	88	67
[5]	122		
[6]	40	40	40
[7]		122	122
[8]		67	88
[9]	94	94	94
[10]			
[11]			
[12]	29	29	29
[13]	63	63	63
[14]			
[15]			
[16]	84	84	84

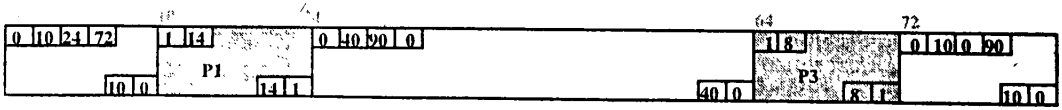
9. Length of the longest chain is 3 and the keys in it are 1457, 8255, 1061

Chapter 10
Storage Management

1. In first fit and worst fit, memory is allocated from 120K free block and in best fit, memory is allocated from 72K free block.

2.





P1 is allocated in a 7-block starting at 0, P2 is allocated in a 5-block starting at 128
P3 is allocated in a 6-block starting at 192, P4 is allocated in a 6 block starting at 256
P5 is allocated in a 7-block starting at 384

The address 256 in binary is 100000000 and the 5^{th} bit is 0, so the 5-block at 256 has a right buddy and the address of the right buddy is 288(100100000).

The free 5-block at 256 is combined with its right buddy to from a free 6-block at 256 which is combined with its right buddy to form a free 7-block at 256. The free 7-block at 256 is combined with its right buddy to form a free block 8-block at 256.

P1 is allocated in a 7-block starting at 123, P2 is allocated in a 8-block starting at 89,
P3 is allocated in a 7-block starting at 55, P4 is allocated in a 8-block starting at 0

The LBC of block P3 is 0, i.e. it is a right buddy. Its left buddy would be a 9-block at address 0 (55 - 55). The left buddy is free so both are combined to form a free 10-block. The LBC of combined free block will be 1 which implies that it is a left buddy. Its right buddy would be a 9 block at address (0+89). But the block at address 89 is not a 9-block which means that right buddy is not available.

INDEX

A

Abstract data types, 1
Acyclic graph, 327
Address Calculation Sort, 462
Adjacency, 327
Adjacency List, 339
Adjacency Matrix, 335
Algorithms, 4
Analysis of algorithms, 4
Array of Structures, 34
Arrays, 10
Articulation point, 330
AVL Tree. 225
  Deletion in AVL tree, 247
  Insertion in an AVL tree, 231
  Left Rotation, 229
  Right Rotation, 227
  Searching and Traversal in AVL tree, 227

■

B-tree, 293
  Deletion in B-tree, 297,311
  Insertion in B-tree, 295,305
  Searching in B-tree, 294
B+ tree, 318
  Deletion, 320
  Insertion, 319
  Searching, 319
Backtracking, 4
Base conversion, 156
Bellman Ford Algorithm, 387
Best Fit method, 493
Biconnected components, 332
Biconnected graph, 331
Binary Buddy System, 498
Binary Search, 473
Binary Search Tree, 202
  Deletion in a Binary Search Tree, 208
  Finding nodes with Minimum and Maximum key, 205
  Insertion in a Binary Search Tree, 205
  Searching in a Binary Search Tree, 203
  Traversal in Binary Search Tree, 203
Binary Tree, 178
Binary tree sort, 451
Bottom Up Merge Sort, 439
Boundary tag method, 495
Breadth First Search, 353
Bridge, 330
Bubble Sort, 424
Bucket Hashing, 490
Buddy Systems, 498

C

Call by reference, 23
Call by value, 23
calloc( ), 29
Circular linked list, 72
  Creation of circular linked list, 77
  Deletion in circular linked list, 77
  Insertion in a circular Linked List, 75
  Traversal in circular linked list, 74
Circular Queue, 122
Collision Resolution, 480
Compaction, 506
Complete Binary Tree, 182
Complete graph, 329
Concatenation of linked lists, 97
Connected Components, 330
Connected Graph, 329
Connectivity in Directed Graphs, 332
Connectivity in Undirected Graph, 329
Cycle, 327
Cyclic graph, 327

D

DAG, 327
Data structures, 2
Data Type, 1
Degree, 328
Depth First Search, 364
Deque, 126
Digital Search Trees, 321
Dijkstra's Algorithm, 378
Diminishing Increment Sort, 431
Direct Recursion, 169
Directed Graph, 326
Divide and conquer algorithm, 4
Double Hashing, 482
Doubly linked list, 63
  Creation of List, 69
  Deletion from doubly linked list, 69
  Insertion in a doubly linked List, 65
  Reversing a doubly linked list, 72
  Traversing a doubly linked List, 65
Dynamic Memory Allocation, 27

E

Expression tree, 201
Extended Binary Tree, 181

F

Factorial, 150
Fibonacci Buddy System, 502

ibonacci Series, 158
irst Fit, 493
loyd's Algorithm, 392
orest, 177, 334
ragmentation, 494
ee( ), 30
reeing memory, 494
ull binary tree, 182
ully in-threaded binary tree, 215
unction calls, 149
unction Returning Pointer, 25

G

arbage collection, 506
eneral Tree, 291
raph, 326
reatest Common Divisor, 158
reedy algorithm, 4

H

ashing, 476
ash functions, 478
Division Method(Modulo-Divison), 479
Folding Method, 479
Midsquare Method, 479
Truncation (or Extraction), 478
eap, 277
Building a heap, 284
Deletion, 282
Insertion in Heap, 279
eap Sort, 455
eight of Binary tree, 200
uffman Codes, 289
uffman Tree, 287

I

place Sort, 420
cidence, 327
degree, 328
direct Recursion, 169
direct Sort, 419
order Traversal, 190
put restricted deque, 129
sertion Sort, 427
olated vertex, 329

K

ruskal's Algorithm, 405

Left in-threaded binary tree, 215
Level order traversal, 194
Linear data structures, 3
Linear Probing, 480
Linear search, 472
Linked List, 48
Linked List with Header Node, 79

M

malloc( ), 28
Mark and Sweep, 507
Merge Sort, 434
Merge Sort for linked List, 441
Merging, 435
Merging of linked lists, 93
Minimum spanning tree, 398
Modified Warshall's Algorithm, 392
Multigraph, 329
Multiple edges, 329
Multiway search tree, 292

N

Natural Merge Sort, 444
Nested Structures, 36
Non linear data structures, 3
Null graph, 329

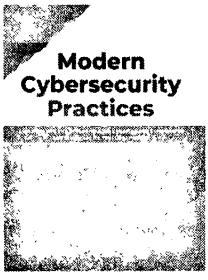
O

O notation, 6
One Dimensional Array, 10
Open Addressing (Closed Hashing), 480
Outdegree, 328
Output restricted deque, 129

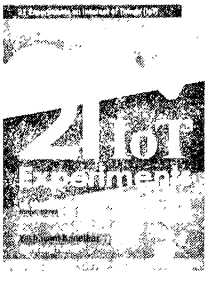
P

Path, 327
Path Matrix, 346
Pendant vertex, 328
Planar graph, 329
Pointer to Pointer, 20
Pointers, 18
Pointers and Functions, 23
Pointers and One Dimensional Arrays, 21
Pointers to Structures, 38
Pointers within Structures, 39
Polish Notation, 137
  Converting infix expression to postfix expression using stack, 139
  Evaluation of postfix expression using stack, 141
Polynomial arithmetic with linked list, 98
  Addition of 2 polynomials, 101
  Creation of polynomial linked list, 101

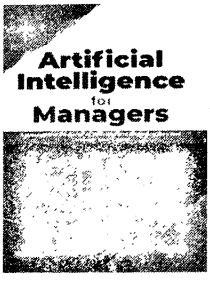
TITLES OF INTEREST



Modern Cybersecurity Practices
Author: Pascal Ackerman
ISBN: 9789389328257

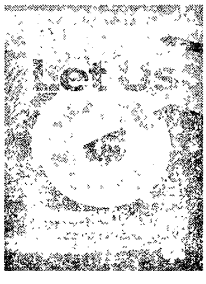


21 Internet Of Things (IOT) Experiments
Author: Kanetkar/ Korde
ISBN: 9789386551832

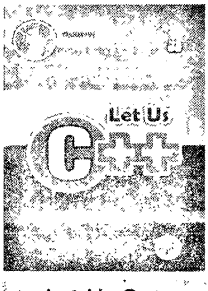


Artificial Intelligence for Managers
Author: Malay A. Upadhyay
ISBN: 9789389898385

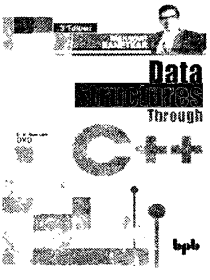
Let Us **Java**
Author: Yashavant Kanetkar
ISBN: 9789388176385



Let Us **C**
Author: Yashavant Kanetkar
ISBN: 9789389845686

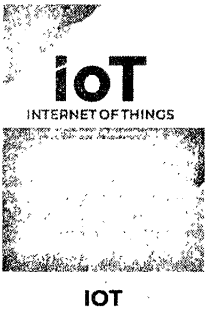


Let Us **C++**
Author: Yashavant Kanetkar
ISBN: 9789388176644

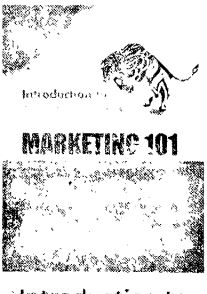


Data Structures Through **C++**
Author: Yashavant Kanetkar
ISBN: 9789388511360

Let Us Python
Author: Kanetkar/ Kanetkar
ISBN: 9789389845006

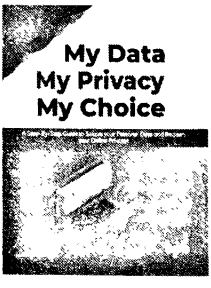


IOT
Internet of Things
Author: Lakhwani/ Gianey/ Wireko/ Hiran
ISBN: 9789389423365



Introduction to **Digital Marketing** 101
Author: Cecilia Figueroa
ISBN: 9789389328189

Python for Developers
Author: Mohit Raj
ISBN: 9788194401872



My Data My Privacy **My Choice**
Author: Rohit Srivastw
ISBN: 9789389845181

Available on **www.bpbonline.com** and all leading book stores.

PUBLISHER OF COMPUTER BOOKS

YEARS OF EXCELLENCE

MILLION BOOKS SOLD WORLDWIDE

In Easy Steps Series

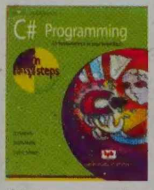
for smart learning

Want to master the latest technology, improve skills? Then, In Easy Steps series is ideal for students. It is easy to follow, having jargon-free content for fast and productive learning.

Features of IN EASY STEP Books :-

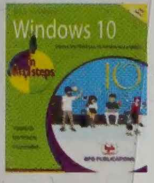
- Over **5 Million copies sold**
- **Easy to follow** and
- Fully **Illustrated**
- Master the
- **Perfect**
- Coveri
- Ea
- E
- in the bo

BPB PUBLICATIONS
DATA STRUCTURES THROUGH C IN DEPTH
Second Revised & Updated Edition
BPB PUBLICATIONS
©2018 BY
ISBN
978-81-7656-741-1
81-7656-741-6
THIS CD-ROM IS FREE WITH THIS BOOK HAS NO COMMERCIAL VALUE AND CANNOT BE SOLD SEPARATELY.
NO LICENCE OR SUPPORT INCLUDED
ALL RIGHTS RESERVED. PUBLISHED BY : BPB PUBLICATIONS, MUNISH PLAZA, 20, ANSARI ROAD, DARYA GANJ, NEW DELHI-110002.
Web : www.bpbonline.com
E-mail : bpb@vsnl.com
Ph : 23254990/91

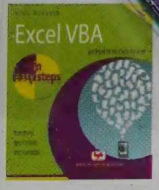


C# Programming

C++ Progra



Windows 10



Excel VBA

Excel 2016

PHP & My SQL

Photoshop Tips, Tricks & Shortcuts

R for Data Analysis

Java

www.bpbonline.com

bpb PUBLICATIONS | #1 Publisher of Computer Books | 62 Years of Excellence | OVER 90 Million Books Sold Worldwide

DATA STRUCTURES THROUGH C IN DEPTH

Second Revised & Updated Edition

"Data structures through C in Depth" presents the concepts of data structures in a very clear and understandable manner. It covers data structures syllabus of different undergraduate and post graduate courses and can be useful for both beginners and professional programmers. This book can be used by students for self study as the concepts are explained in step-by-step manner followed by clear and easy to comprehend complete programs. The explanations are illustrated by detailed examples, figures and tables throughout the book. Exercises with solutions are provided which help in having a better understanding of the text. The CD-Rom contains all the programs given in the book. Some 'demo' programs are included in the CD, which demonstrate the stepwise working of the algorithms.

- **Introduction**
- **Arrays, Pointers and Structures**
- **Linked Lists**
- **Stacks and Queues**
- **Recursion**
- **Trees**
- **Graphs**
- **Sorting**
- **Searching and Hashing**
- **Storage Management**

About the Authors

Suresh Kumar Srivastava has been working in software industry for last 12 years. He has done B level from DOEACC Society. He likes to work on system side and do some creative work for development of software tools. He has authored a book on C language titled "C in Depth".

Deepali Srivastava has done M.Sc. in Mathematics and Advanced PGDCA from MJP Rohilkhand University. Her areas of interest are C, C++ and Data Structures, and she has a passion for writing. She has authored a book on C language titled "C in Depth".

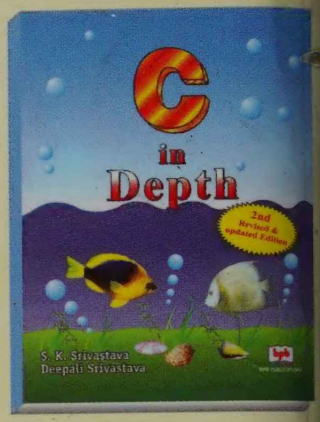
ISBN - 13 : 978-81-7656-741-1
ISBN - 10 : 81-7656-741-8

www.bpbonline.com

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj, New Delhi-110002

Rs.390/-

Other Title of Interest :



C in Depth

2nd Revised & Updated Edition

Pages : 552

ISBN : 81-8333-048-7

The book explains each topic in depth without compromising over the lucidity of the text and programs. This approach makes this book suitable for both novices and advanced programmers. The well - structured programs are easily understandable by the beginners and useful for the experienced programmers. The book contains about 300 programs, 210 exercises and 80 programming exercises with solutions of exercises and hints to solve programming exercises. The chapter on project development and library creation can help students in implementing their knowledge and become a perfect C programmer.

ISBN 81-7656-741-8
9 788176 567411